# TAXONOMIES FOR SOFTWARE SECURITY

by

HORIA V. CORCALCIUC

A thesis submitted to
University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

School of Computer Science
Engineering and Physical Sciences
University of Birmingham
November 2013

## Abstract

A reoccurring problem with software security is that programmers are encouraged to reason about correctness either at code-level or at the design level, while attacks often tend to take places on intermediary layers of abstraction. It may happen that the code itself may seem correct and secure as long as its functionality has been demonstrated - for example, by showing that some invariant has been maintained. However, from a high-level perspective, one can observe that parallel executing processes can be seen as one single large program consisting of smaller components that work together in order to accomplish a task and that, for the duration of that interaction, several smaller invariants have to be maintained. It is frequently the case that an attacker manages to subvert the behavior of a program in case the invariants for intermediary steps can be invalidated. Such invariants become difficult to track, especially when the programmer does not explicitly have security in mind. This thesis explores the mechanisms of concurrent interaction between concurrent processes and tries to bring some order to synchronization by studying attack patterns, not only at code level, but also from the perspective of abstract programming concepts.

*“I had been hungry all the years-*

*My noon had come, to dine-*

*I, trembling, drew the table near*

*And touched the curious wine.”*

*I Had Been Hungry All the Years* **- Emily Dickinson**

# ACKNOWLEDGEMENTS

I would like to thank my PhD supervisor Hayo Thielecke for accepting to be the promotor for my thesis. His continued patience, outstanding knowledge and our meetings have been essential to the completion of my thesis as well as contributing greatly to my general development.

I would also like to thank Mark Ryan the continued guidance and selfless support that he has granted me during all my years of study. Aside from recommending me as a Teaching Assistant, he has been available and kind to me even during the harshest of times.

Last but not least, I would like to thank my parents for the enduring support that they have shown me. Without their constant guidance, help and understanding this thesis could not have been completed. They have invested a lot of effort to keep me on track and to support my choice of studying for a PhD.

# CONTENTS

# LIST OF FIGURES

CHAPTER 1

# INTRODUCTION

When dealing with security in a concurrent context, one expects heavyweight designs based on intricate forms of inter-process communication (IPC). Looking at mutual exclusion locks (mutexes), semaphores, resource locks and shared memory, all of them are elaborate designs of IPC which are created with the purpose of passing data rather than providing a simple means of synchronization. Depending on the complexity of the information to be exchanged, programmers have to deal with unpredictable events that could pose a security risk. It is easy, albeit tedious, to perform a static analysis of non-asynchronous programs but considerably more difficult to reason about security and code safety in a concurrent context. One outstanding conclusion that we are able to draw is that the more interleaved processes are, the more likely it is to come across some unpredictable events. To avoid unpredictable events, the most trivial solution would be to minimize the interaction between processes. There are several ways to do that, for example, one could minimize the number of instances where processes are meant to communicate or one could minimize the amount of exchanged data. In the numerous case-examples we have gathered during this thesis, it is interesting to observe that vulnerabilities are more frequently based on abstract, low-level concepts that would require a redesign of the programs involved. In order to build a taxonomy for concurrency-based attacks, we had to investigate intermediary design levels of programs in order to find universal patterns that can be applied to a wide palette of attacks.

This has lead us to UNIX systems and to explore signals and signal handlers in order to gain insight on how IPC was historically reasoned about. We found that signals are a lightweight mechanism that allows us to pass short messages between processes in order to notify a process that a certain event has occurred. In order to do that, a process installs a handler and waits for a signal to be delivered. When the signal is processed, the execution of the main thread stops and the signal handler runs. There are a number of signal identifiers for every type of signal that a process can send or receive. When a process receives a signal, execution stops and the process is instructed to execute the commands within the signal handler block. The signal handler does not run concurrently with the main thread but the main thread can be interrupted at any time so that the handler may run. After the execution of the signal handler control flow returns to the place where the signal was originally delivered to the process. Signals and signal handler mechanisms are particularly interesting because a program can be interrupted at any point which puts a lot of pressure on a programmer to make sure that the program is ready to be interrupted. We have seen an impressive amount of attacks which are based on the fact that an interruption of a program at an inappropriate time can be abused at the level of control flow so that execution is diverged to a state that is favorable to an attacker. There are several concepts that are centered around interruptions that have strong ties to fundamental security concepts such as denial of service attacks, time-of-check-to-time-of-use (TOCTTOU), privilege escalations and even arbitrary execution. The most severe cases, we have found, appear after a staging attack that is based on diverging control flow at an inappropriate time.

While studying signals, we have determined that the connections between signal-based attacks are subtle. For example, a signal can be received at any time after the process has bound the signal identifier to a signal handler. From the perspective of control flow, this leads to the interleaving between the signaling process and the process waiting for the signal to be delivered. We can identify another subtle layer of concurrent interleaving that takes place within the same process. Due to the fact that an interruption may occur at an inappropriate time, if the program is not carefully designed to handle the interrupt, then some part of state may become corrupted. It is difficult to reason about concurrent processes and this research has focused on concurrency by studying the interaction of processes at the level of flow control. Various tools are used, such as the notion of traces, rely and guarantee-logic and stability. We use traces extensively in order to reason about the interleaving of processes with the hope that some bad behavior, due to the concurrent interleaving of processes, may be factored out, while having in mind to not violate too many dependencies. Rely and guarantee is used, which allows us to specify sets of restrictions that we can bind to the traces involved in the interleaving. This coupling of traces and rely and guarantee allows us to reason about exploits and provides a separate specification, for each case, which we later generalize and provide a template so that security flaws can be avoided in the future.

## 1.1 Motivation

Going further we have noticed that some signals implementations provide a flag that allows a programmer to make a signal handlers run only once, which we name one-shot signal handlers. If the signal handler is declared to be a one-shot signal handler then it can run only once so that a second pass through the signal handler is not possible. There are a certain similarities between one-shot signal handlers and exceptions mainly due to the fact that they can both be used to discard entire blocks of operations that are designed to run only once. A typical example of such operations, would be a cleanup functions that contains asynchronous unsafe functions. A function that de-allocates memory is not safe to be called more than once because a subsequent call would attempt to release the same resources a second time. These functions are tagged as being non-reentrant, meaning that a subsequent call should not be attempted.

Our research was motivated by an attack published by Zalewski[89] in Phrack, as a case study on the Sendmail daemon. Zalewski provides several examples of non re-entrant safe functions and illustrates how a process may be interrupted by delivering a signal at an inappropriate time. This is a solid example where a one-shot signal handler could be put to use. In order to eliminate these commands we reason about control flow using traces [23] and identify the dependencies between the program and the environment. If it is the case that factoring out certain commands eliminates some unwanted behavior, and if that change does not alter the meaning of the program too much, then they may be removed in order to fix a design flaw.

We have noticed that reasoning with traces is applicable to other attacks, not necessarily related to signals. For example, Watson [83] describes an exploit on a shared-memory segment that can be reasoned about elegantly by making use of traces and stability. As Watson's paper describes the attack, a memory buffer is overwritten by an attacker between the time of check and the time of use. This leads us further to rely-guarantee logic which allows us to define a set of conditions that a well-behaved program will have to respect in order to be considered safe. In Watson's case we can impose a restriction on the shared-memory segment so that predicates are not abused between the time of check and the time of use. This makes sure that the address is not overwritten due to an interfering attacker. In doing so, we introduce the concept of stability which allows us to define what must not change between the states of a program.

We use Aczel traces from de Roever et al. [23] which allows us to observe the concurrent interleaving of programs as a sequence of traces where each program or thread contributes with one or more commands. From a control-flow perspective, the composition of traces merges programs together so that every postcondition can be seen as the result of the individual contribution of each implicated thread. In that sense, two different programs that depend on each other may appear as one single program to an external observer from the perspective of control flow. The Aczel trace model describes a tuple consisting of two states, a begin-state and an ending state as well as a process identifier that marks processes that a particular trace belongs to. The original Aczel model contained a process identifier that is part of a trace and allows every process to be tracked at every single state-change. We have removed the process identifier because it does not tell us too much about the interaction between processes when applying the model to security. This is mainly due to the fact that security flaws usually involve a limited amount of participants so that determining which process is the culprit becomes trivial. We are also to specify conditions that traces that restrict certain actions during a transition from a precondition to a postcondition. A trace relies on some environmental condition, for example a thread could depend on the fact that some shared resource is accessible. A trace also offers its own set of guarantees to the environment which makes sure that it does not violate the dependencies other processes may have. A mutual relationship is established, each trace offering a set of guarantees to the environment and in-turn each trace relies on the environment. Although this would appear as circular reasoning, that is not the case because the traces do not guarantee, respectively rely on the same set of conditions. The relationship can be seen as an agreement between processes where each involved process relies on something that it needs from the environment and additionally offers something *different* in return.

Trace compositions, predicates and stability allows us to model the interleaving between parallel processes by specifying sets of relies and guarantees that give us an overview of control flow. This allows us to classify attacks and build a taxonomy based on similarities between vulnerabilities.

Initially this research began by studying separation logic [10] in order to reason about memory vulnerabilities where the heap must be split up. For the taxonomy though, we have used regular disjunctions to separate predicates. As further work, traces allow us to extend the logic and combine rely and guarantee with separation logic [80]. For the purpose of building a taxonomy, we have been more interested about reasoning using traces and studying programs from the perspective of control flow.

## 1.2 Overview

We first offer an overview of the various tools that have been used for security and describe the concepts behind them. This part is covered in the literature overview in the "Research Context" and "Secure Languages and Tools" . In the "Flavors of Logic" chapter, we revise the various flavors of logic that offer insight on how to deal with concurrency where security and program safety is cucial.

In the "Signals and Events", we describe the implementation of signals and observe that a split between one-shot and persistent signal handlers is available in the implementation. Based on these functions, we then explain in detail how the signaling mechanism is designed.

The centerpiece of the thesis consists of a taxonomy, in the "A Taxonomy on Layers of Abstraction" chapter, that summarizes all the literature overview and offers taxonomy trees that allow us to classify vulnerabilities using shared properties. We expand the concept of traces and use swimlane diagrams to depict traces pictorially, thereby offering a graphical overview of control flow. We reason about vulnerabilities but we also show measures that could be taken in order to prevent attacks that follow similar and related attack patterns.

Additionally, since we classify Denial of Service (DoS) as a self-standing taxonomy tree, we identify three types of classic DoS attacks that abuse programs in different ways. We use automata to describe the attacks and make the connection with swimlane diagrams that we mention in the former chapters on taxonomies.

The last chapter contains conclusions and further directions that could be followed in order to expand the taxonomy that we provide to a broader palette of attacks.

## 1.3 Research Context

The problem with signals is that there is no theoretical framework built that would clarify in what particular scenarios they are meant to be used. We noticed that system daemons were the main software packages that made use of signals and signal handlers. We turned to programming language design to explore the security implications that signals could have.

In order to study concurrency in a security context, we have gathered exploits that target programming oversights. An exploit can be described as taking advantage of a vulnerable software package. The usual intention of an exploit is to gain access to a system, or disrupt a service so that it is rendered unusable. The vulnerabilities on which these exploits rely can be roughly classified as being one of the following types:

- Time and state attacks
- Denial of service attacks
- Format string attacks
- Code injections
- Race conditions
- Overflows

  Buffer, heap, stack overflows

  Integer overflows

The divisions and subdivisions are a general overview and they can be seen as containers for different variations on the same general outline which provides the necessary motivation for a unified classification such as the taxonomy of "Time and State" attacks.

One substantial subdivision consists in buffer overflows which were introduced and described in detail by "Aleph One" in Phrack [19]. Buffer overflows rely on the shortcomings of some languages, such as C, that do not allow enforcing a policy for memory access. Managing memory is left as a task that a programmer must accomplish with no help from the compiler. In certain cases, this sort of low level access may be preferential because raw memory access allows developers to write efficient programs. A good example could be a platform with low resources where a program could be optimized in order to deal with the constraints of a limited stack. However, the same freedom also allows programmers to commit errors and write software that may be vulnerable. Buffer overflows are considered to be one of the most widespread attacks, since the release of the Morris worm in 1988 [62].

Consider the following code in Figure 1.1 which illustrates a short program vulnerable to a buffer overflow attack.

```
void f(char *data) {
char s[10];
/* No bounds check done */
strcat(s, data);
}
void main(int argc, char *argv)
{
  f(argv[1]);
}
```

**Figure 1.1:** No bounds-checking is performed which would check whether the character array `s` can hold all the data that is copied from the memory referenced by the pointer `data`.

When the program runs with certain command line parameters, the program passes the command line parameters to the function `f` where they are copied into the memory block pointed to by the character pointer `s`. The pointer `s` references a memory block on the stack that can only hold `10` characters. If the command line arguments exceed this length then the rest of the memory, past the ten characters bound will be overwritten by the rest of the data from the command line arguments. Suppose the figure below is an illustration of the stack before and after the attack. By debugging the program and by passing identifiable characters to the program on the command line the attacker can find the location of the return address of the function `f`. Then by overwriting the stack and the return address the attacker is able to make the program jump to any memory location. Frequently, the slice of stack before the return address is overwritten with opcode commands that perform some action favorable to the attacker - for example, opening a shell, as it seems to be the most frequent case. The injected code can be anything as long as it would fit the storage between the buffer and the return address. In case the injected code is smaller, the rest of the stack up to the return address may be padded by NOPs, thereby making the program slide to the return address (Figure 1.2).

If the injected code would fork a command shell, then that shell would run with the same permissions of the running program. As a consequence, it is commonly suggested to run system services with low user-permissions in order to limit the capabilities of potential attackers.

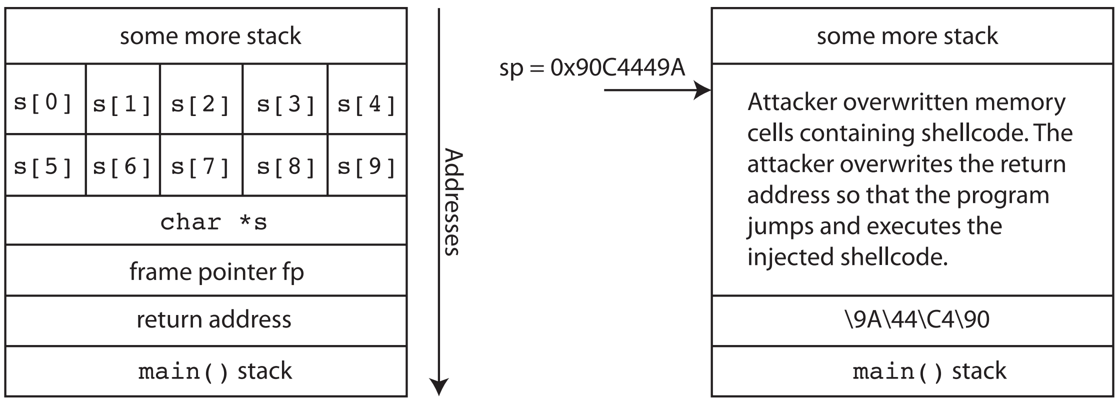


**Figure 1.2:** Writing to memory directly and overwriting the return address in order to execute arbitrary code.

Resource sharing between threads has been a pending issue because specifying precisely what resources may be accessed, at different times, by several concurrent processes is difficult to reason about. A good example would be two processes that access the same file and write a certain amount of data to that file. Without any policy, it is not clear what contents the file will have after the two programs run. In fact, even for the duration of the file writing operations, one does not know what data the file holds at a certain time. We would not know how to determine the order in which the write operations occur or whether one program will succeed in writing its data before the other program terminates. Such problems broadly fall into the category of race conditions, based on the observation that both programs race each other to dump their data to the shared file.

There are two concepts that apply in such cases:

- Thread safety
- Re-entrance safety

Thread safe functions perform actions atomically, within one clock cycle and do not allow other operations to interfere with the data that they handle. On many platforms, processes are not able to transfer data to storage instantly. As an example, processes may write to a file simultaneously, thereby interleaving and overwriting each other's data. As stated by the GNU C library manual page, it is feasible to generally assume that file-writing operations are not atomic. In practice, very few functions and operations are atomic.

On the other hand, re-entrance safety does not relate to atomicity, but rather to the fact that some shared resource should not be manipulated in a certain way more than once. The Zalewski exploit discloses a vulnerability within the Sendmail daemon. A memory segment is referenced by a pointer and freed once by a signal handler. If the signal handler would run after the memory is deallocated, the same memory location will be freed up. This would lead to undefined behavior and the outcome would depend on the system implementation. In may happen that the second attempt to free the same memory segment would fail silently or, on other systems, it could result in the whole program crashing.

There are several conditions laid down by the GNU C library manual which specifies under what circumstances a function may be considered re-entry safe [28] as explained by at IBM [47]:

- The function must not hold any static data over successive calls.
- The function must not return a reference to a static non-constant data.
- The function must protect global data by making a local copy of it.

- The function not call non re-entrant functions.

Going back to the simultaneous file-write example, we can see that input-output (IO) related functions rely on shared singleton resources, for example hard storage that is shared amongst all programs. This definition narrows down a whole set of functions to very few that may be considered re-entrant safe. Following the guidelines, it is sometimes feasible to rewrite functions in order to turn them into re-entrant safe functions. For example, looking at the first rule, we could eliminate commands which return pointers to data defined statically within the function body as illustrated in Figure 1.3.

```
char *global_pointer;
void f() {
  free(global_pointer);
}

int main(void) {
  f();
  f();
}
```

```
char *global_pointer;
int flag=0;
void f() {
  if(flag==0) {
    free(global_pointer);
    flag++;
  }
}
int main(void) {
  f();
  f();
}
```

**Figure 1.3:** De-allocation of a pointer using a non-reentrant function `f` on the left in contrast with a re-entrant safe function `f` on the right. The memory allocation for `global_pointer` has been left out for brevity.

The main function would do two subsequent calls to the `f` function that would result in a double free corruption. The `f` function operates on the pointer `global_pointer` thus making the function `f` a non-reentrant safe function. In this simple case, we can find a way around it by simply defining a global variable.

This modification on the right-hand side in Figure 1.3 would make the code re-entrant safe since the flag would be incremented after freeing the memory referenced by the global pointer, and a second de-allocation, upon re-entry of the function `f`, could be prevented by a test condition. On the other hand, it is important to note that the test `flag==0` is not an atomic operation and if two threads were to execute the same function simultaneously, the test may fail twice and `free()` will be called twice by two different threads which will not eliminate the double-free issue. The usual solution when dealing with threads that access resources concurrently is to use a locking mechanism instead.

There are certain interesting things to note about this example. One can observe that the `flag==0` test itself is still required because, once the lock is released, the other thread may acquire the lock and then proceed through the code. Without the `flag==0` test, the second call of `free()` would again lead to a double-free error, even if mutex locks have been used to grant exclusive access to the code-block.

We have shown that this type of vulnerability is based on two distinct concepts: thread-safety and re-entrance safety. In many cases, both properties have to be used in order to ensure the safety of a program such as the one given in this example.

CHAPTER 2

# SECURE LANGUAGES AND TOOLS

Certain dialects of C such as Cyclone[22] or Vault[24] implement mechanisms for program safety at the language level. These dialects include protections including bounds checking and regions in order to prevent overwriting of memory segments that are not bound to a particular code-block. They borrow concepts from Hoare logic [43] and separation logic [67], sometimes even making logical constructs such as pre- and postconditions explicit within the language.

As an example Vault makes logic reasoning explicit by implementing user-definable states. The Vault project at Microsoft Research [35] is centered around controlling memory and lifespan of user-defined variables and states. As a heavily modified dialect of C, Vault contains function prototypes and introduces keywords such as the "tracked"-keyword which allow the compiler to handle those resources more carefully. A function can thus create resources, dispose of resources and change the states associated with these resources. Figure 2.1 shows function prototypes that are an example of how this is accomplished.

```

tracked($F)FILE open read(string)[new $F @ readable J];
void write(tracked($F)FILE,char)[$F @ writable J];

```

**Figure 2.1:** The function prototypes use annotations that mark certain variables or functions as being tracked or untracked.

In the example above $ is the symbol for key. Vault uses keys to reference resources. Each resource has an associated key which can be in a certain state. We can observe the Hoare-style reasoning since the explicit states can be seen as pre- and postconditions. When the compiler performs a static analysis, it alleviates the programmer's job because it becomes easy to spot whether a "readable" variable or a "writable" limitation imposed on a resource has been violated. Trivially, one could not write to a resource that has been declared as being "readable" but not "writeable".

Cyclone also implements something called "region handling", which is perhaps the first predecessor of the regions that we now find in object-oriented programs such as C#. Cyclone allows to define a region [30] as being a subpart of the heap containing only the objects that are allocated in that part of the heap. After use, the whole region along with all the objects are disposed of entirely which makes common oversights such as dangling pointers and leaks easier to fix. Also, by allowing the isolation of memory within regions, buffer overflows become hard to accomplish since the only memory that an attacker could overwrite is isolated within a slice of the heap.

The Microsoft Research website gives a lot of pointers on how to use Vault, although the projects has been abandoned in favor of a new project called "Singularity" [34]. This new system imports concepts from Vault but expands on them by using a new language called Sing# [45] which is a rough offspring of C#.

Other papers from Microsoft Research mention a tool called "Smallfoot" [6] is an advanced static checker as well as other similar checkers intended for particular usage - such as verifying the code-safety of device drivers.

## 2.1 Vault

Vault is considered to be a safe dialect of the C programming language developed at Microsoft Research. It provides the same level of safety as C# but additionally provides the programmer with a fine-grain control over the lifetime and layout of data. Unlike

Cyclone, which is geared towards security, Vault is built with code safety in mind. For example, most of the distinctiveness of the dialect can be observed on a higher level and focuses on function calls, variants, aggregate types as well as borrowing some object oriented concepts, such as generics. Vault is best seen as redesign of the C language, unlike Cyclone, which just adds constructs that are meant to deal with security and code safety. The general impression we get from Vault semantics is that the dialect focuses on implementing data access control, an off-spin of access control lists, focused on resources, with little support for the vulnerabilities that Cyclone addresses. More precisely, Vault adds programming logic, observed at the level of syntax, that allows developers to be precise about accessing variables. On the other hand, it seems that adding data access control is largely redundant since it requires the programmer to have a well-defined image of the code to be written, which still leaves opportunities to make mistakes.

Vault implements modules and interfaces, borrowed from object-oriented programming. Modules are a collection of type, variable and function definitions that can be seen as a generalization of C-like structures. One can declare modules to be inner or outer resembling protection levels of simple class objects. Encapsulation can be done by using the static keyword when declaring variables or functions inside the module. Interfaces provide encapsulation and information largely similar to a C-style headers. Similar to Java, an interface acts as a contract between a module implementation an a module client. Dynamically, a module can claim or adopt an interface in order to implement it.

Arguably, because of the many object-oriented inspired additions, Vault appears more like Java than C. It implements objects, complex scoping and all the main characteristics one would find in an object oriented programming language. Vault is also most likely a Microsoft predecessor of the C#-programming language where the connection between the low-level memory access of C and the object oriented C#-programming language becomes quite shallow: the only borrowed concepts being mainly the syntax.

## 2.2 Cyclone

Cyclone is a safe dialect of the C programming language designed to prevent buffer overflows and is geared towards security rather than code safety. Cyclone launched as a collaborative project between AT&T Labs Research and Greg Morrisett's group in 2001. The first version of the language was released on 8th of May 2006. Regrettably, since 2006 there have been no major updates to the project and development has ceased. Using a couple of restrictions, Cyclone claims to maintain the look and performance of C programs by adding just a few constructs which do not stray away too far from the overall aspect of the original C syntax:

- `NULL` checks are inserted to prevent segmentation faults for normal `*`-type pointers.
- Pointer arithmetic is restricted and pointers are split up into various other subtypes (those supporting pointer arithmetic and bounds checking like fat pointers, those which are never `NULL` and don't require `NULL` checks such as `?`-pointers and regular `*`-type pointers).
- Pointers must be initialised before use.
- Cyclone implements regions to deal with dangling pointers. `Only`safe casts and unions are allowed (transitions from a certain pointer type to another are restricted).
- `goto` is disallowed into scopes.
- `switch` labels in different scopes are disallowed.
- Functions returning pointers must execute return.
- `setjmp`, `longjmp` and more generally concurrency are not supported (however Cyclone allows thread libraries and agrees to compile yet the resulting program does not create any threads, they are just there in the form of stubs).
- arrays and strings are automatically converted to `?`-pointers.

```
/* Standard C Variant */
int strlen(const char *s) {
  int i = 0;
  if(!s) return 0;
  while(*s) i++;
  return i;
}
```

```
/* Cyclone Variant */
int strlen(const char ?s) {
  int i, n;
  if(!s) return 0;
  n = s.size;
  for(i=0; i<n; i++, s++)
    if(!*s) return i;
  return n;
}
```

**Figure 2.2:** The `strlen` function in both plain C and the Cyclone dialect side-by-side.

Cyclone also implements a few extensions to the C programming language bringing, like Vault, some elements of object oriented programming languages:

- Garbage collection for heap allocated values.
- Exceptions.
- Polymorphism that replaces some uses of `void *`.

The main strength of Cyclone comes from splitting pointers into three categories.

As we can see in Figure 2.2, for the standard C variant, the loop would be unsafe if `s` would not be `NULL` terminated. The Cyclone version forcibly converts the pointer to a `NULL` terminated pointer and also allows to check the size of `s` using a reference construct. In the plain C example, if `s` would not be `NULL` terminated the result would be undefined behavior (depending on the implementation, a `SIGSEGV` signal may be sent to the application).

```
int fun(int *);
```

```
int fun(int @);
```

**Figure 2.3:** Two function declarations, side-by-side: the left function is defined using C-like unsafe pointers and on the right using Cyclone's fat pointers.

Taking an even simpler example illustrated in Figure 2.3, suppose the following function was written in standard C. If the function `fun` would not include a `NULL` check for the pointer argument, it might again lead to undefined behavior.

This is telling the compiler that the argument of the function fun should never be `NULL`. Of course, it is important to emphasise the fact that this problem could also be avoided in standard C by just including a `NULL` pointer check inside the function which would return (like in the previous example) if the pointer would be found to be `NULL`.

Overall, Cyclone brings in some new features to the C programming language also adding some concepts of object oriented programming languages. It is important to note that these features are not necessarily based on logic but are rather based on intuitive reasoning about issues governing code writing and code safety. In other words, the set of additions try to circumvent common mistakes made by programmers yet they could be avoided altogether by a careful programmer implementing checks or managing memory properly.

## 2.3 CCured

CCured [86] is a source parser that takes plain C code and enhances it by inserting runtime checks to prevent memory misuse similar to CRED [68]. When the transformed source is compiled, CCured claims that it able to prevent all memory violations by halting execution before misusing memory. This comes at the cost of a considerable performance impact yet it can be improved by adding manual checks and by having CCured exclude code sections. CCured introduces five primary new basic pointer types. Amongst the main kinds we find `SAFE` (not used for pointer arithmetic and requires just `NULL`-checks before dereference), `SEQ` (used in pointer arithmetic but no unsafe casts - it requires a bounds checking and a `NULL`-checks), `WILD` (used in pointer arithmetic, unsafe casts and does bounds, `NULL` and dynamic type checks) and some minor ones like `RTTI`, `FSWQ`. At compile time CCured offers suggestions to transform WILD pointers into SAFE and SEQ pointers. When dealing with pointers, the main difference between Cyclone and CCured is that CCured infers pointer types based on usage rather than declaration. An issue common to both Cyclone and CCured is that C-style `vararg` function definitions do not require parameters to have an explicit type which makes the static analysis difficult.

However, CCured does not always produce the desired output and although the developers claim that they have successfully ported many Open-Source projects such as Sendmail, Bind and even Apache modules, CCured requires a lot of manual user intervention. Similar to Cyclone, CCured works by substituting memory related functions, such as `malloc()`, `calloc()` and `free()`. The drawback is that by replacing the memory allocation functions, the programs become much less portable and create an extra dependency on CCured's libraries.

A common complaint that is raised against CCured and Cyclone is that both tools return indecipherable error messages that are not clear, frequently overlap and thereby do not help a developer understand or reason about the fault that occurred.

There are other related tools such as the "C Range Error Detector", CRED [68] that focus on buffer overflows but aside from memory corruption attacks these tools are not able to prevent attacks based on program flow. CCured's strong point is type inference and CRED [68] goes into a detailed analysis of memory bounds. Neither CCured or CRED have any constructs that deal with concurrency.

## 2.4 The GNU Compiler

The GNU Compiler (GCC) offers some minor protections included in the compiler itself and triggered by specific options. We would like to recall some of the options supported in the recent compiler releases since they employ standard and well-known techniques such as honeypots, un-executable stack and address space randomization. Some features, such as the stack guard protector [11] and address space randomization (ASLR) [70] that have been formerly analyzed and defeated. A related operating system-level features comes from GRSEC [64] kernel extensions which randomizes PIDs every time a process runs although that decreases the portability considerably and breaks programs that rely on the sequntial PID selection algorithm.

- Source fortification (triggered by specifying `-DFORTIFY_SOURCE` to the gcc compiler). This is a static check, implementing bounds-checking and is meant to prevent buffer overflows. it is relies on a very simple idea that in certain cases we actually know the length of a buffer such as statically allocated buffers where the size of the buffer on the stack is fixed or if the buffer was just allocated using `malloc()`). If we know the size, it is arguably easy to ensure that those bounds are not overstepped. This is a new feature, to be found in gcc version 4.2 onward.

- SSP [82] (Smash Stack Protection) is another feature which can be triggered by the `-fstack-protector` flag on recent gcc compiler versions. The addition is inspired from the concept of honeypots where certain markers are placed on the stack, which would allow the runtime to detect any tampering with the stack. SSP moves further and prevents memory corruption by reordering local variables and merges pointers in function arguments, moving them to an area preceding local variable buffers [20].

Additionally, Linux benefits from a dynamic library called `libsafe` and more recently `libverify` [77] available from Avaya Labs Research that, once linked to a binary, intercepts common C library function calls that are susceptible to buffer overflows and performs bounds checks in order to make sure that frame pointers and return addresses have not been overwritten. It is similar to SSP mentioned previously but the greatest benefit is that it does not require a recompilation of the program and any binary can benefit from `libsave` and `libverify` by linking to them.

## 2.5 Splint

Splint [53] was formerly named LCLint and is a tool for statically checking C programs. It is derived from the historical tool called "lint" now being part of a broader project which is called LARCH [41]. Splint supports writing stand-alone program specifications in separate files using LARCH interface Language for C (LCL) and then using them to statically check code. Splint addresses several problems:

- `NULL` pointer dereferencing

- Use of unallocated memory
- Type mismatch
- Bad aliasing
- Infinite loops
- Buffer overflows
- Badly implemented macros

Take for instance a typical NULL pointer dereferencing side-by-side with the Splint variant as seen in Figure 2.4

```

char f(char *ptr)                         char f(/*@null@*/ char *ptr)
{                                         {
  return *ptr;                              return *ptr;
}                                         }

```

**Figure 2.4:** The declaration of two functions in both standard C and also annotated on the right with Splint syntax. The difference is that the `null` keyword in the function prototype in the Splint variant hints to the compiler that the pointer may be `NULL`

The notation `/*@null@/` tells Splint that the pointer `ptr` might be `NULL`. If `ptr` is `NULL` then splint would report an error claiming that there may be a dereferencing of a `NULL` pointer.

A good example of "getters" and "setters", similar to the ones we find in C#. Consider the following code and the Splint annotated counterpart can be seen in Figure 2.5.

Splint allows the programmer to annotate parameters with an `/*@in@*/` or an output parameter `/*@out@*/` which hint whether the parameters is meant to be fetched or set by the function. In the previous example, Splint would report a misuse on storage `x` not being correctly defined for `get(x)` and for `huh(x)`.

```c
extern void set(int *x);
extern void get(int *x);
extern int huh(int *x);

int f(int *x, int i)
{
  switch(i)
  {
    case 1:
      return *x;
    case 2:
      return get(x);
    case 3:
      return set(x);
    default:
      return huh(x);
  }
}
```

```c
extern void set(/*@out@*/ int *x);
extern void get(/*@in@*/ int *x);
extern int huh(int *x);

int f(/*@out@*/ int *x, int i)
{
  switch(i)
  {
    case 1:
      return *x;
    case 2:
      return get(x);
    case 3:
      return set(x);
    default:
      return huh(x);
  }
}
```

**Figure 2.5:** An illustration of using ”getters” and ”setters” in Splint.

For accomplishing standard tasks which may lead to errors Splint adds safe library functions that can be used instead of the `glibc` complement of functions. The ability to write annotations and be more precise about variables makes reasoning much easier. The benefit of using static analyzers are majorly beneficial compared to Vault or Cyclone because they do not require nor enforce the use of an alternate syntax and they do not perform runtime checks that may have a performance impact on the program.

Looking at the dereferencing `NULL` pointer example, the programmer must have some prior knowledge about the function `f` and realize that the character pointer `ptr` may be `NULL`. However, if the programmer is aware that the character pointer `ptr` might be `NULL`, then it would be likely that the programmer would already have written a `NULL` pointer check in the first place.

## 2.6 CQUAL

CQUAL [31] is a type-based analysis tool for finding bugs in C programs. It extends the language by allowing user defined type qualifiers which are used in the same way as the standard C type qualifiers, such as `const`, for example. It can note that values are "tainted" or "untainted" which, for system administrators, is similar to Perl's taint checks where all user supplied input is considered malicious unless the programmer explicitly confirms the validity of the provided data. A CQUAL developer is meant to annotate the program in critical places and CQUAL performs qualifier inferences to check whether the annotations are correct.

As an illustration, suppose we define a type quantifier "unchecked" (Figure 2.6) which we use to mark an object that has not been authorized.

```
struct file * $unchecked fp;
```

**Figure 2.6:** Unchecked pointer declarations in CQUAL.

The previous declaration states that the file object `fp` has not been checked which is useful if a certain function expects to find an argument to be checked before it could use that argument. CQUAL performs checks whether the subsequent operations constitute a type violation. For example, the following calls illustrated in Figure 2.7 are considered a type violation by CQUAL.

```
void f(struct file *$checked fp);
void g(void)
{
        struct file * $unchecked fp;
        f(fp);
}
```

**Figure 2.7:** A type violation in CQUAL: the function parameter `fp` in function `f` expects a `checked` pointer type.

The function `f` expects a checked file pointer as parameter but the passed pointer is of type unchecked. This works since CQUAL also introduces the notion of lattices which, for our example illustrated in Figure 2.8.

```
partial order {
  $checked < $unchecked
}
```

**Figure 2.8:** Lattices in CQUAL.

The example says that `$checked` is a subtype of `$unchecked`. Generally, a lattice is a partially ordered set in which all nonempty finite subsets have a least upper bound and a greatest lower bound. In this particular example, the lower bound would be considered to to `$checked` and the upper bound would be `$unchecked`. This also has the property that a `$checked` type can be used wherever an `$unchecked` type is expected but any reversal would result in a type violation.

A derivate of CQUAL has also been used to find Y2K bugs in C programs. CQUAL has also been used to test [90] the implementation of LSM modules [88].

## 2.7 UML and UMLsec

The Unified Modeling Language (UML) [52, 33] is a graphical language that helps model, document, visualize and specify software systems. The language is not restricted to any level and there are many extensions that take basic rules and expand on them in order to offer tools for modeling more complex processes such as industrial processes or business workflows. UML can be used in any case whenever we need to define a clear and structured workflow where some events take place in a certain order. Loosely, one can see UML as a visualization tool based on the consequences of a Turing machine where certain events appear on the outside as a sequence of steps, each step being constrained by certain conditions. Based on the workflow principle, UML is flexible enough to accommodate any interaction with any number of participants. In software engineering, where processes may appear to execute code in parallel, the concurrent interleaving can still be seen as a composition where each process contributes with actions that have an impact on the overall outcome of a program.

UMLSec [49, 48] is an extension to UML that extends the language so that it supports constructs and constraints that are essential to security. For example, given a message that has to be exchanged over the Internet, the communication can be tagged as occurring over the Internet which could make the message susceptible to tampering during the transaction. Similarly, every prototype define by Jürjens et al. adds constructs which are useful to tag visual diagrams that are known to conform to specific rules. Using these tags, the security of a system can be analyzed visually and at-a-glance by following the rules and constraints imposed on the various components of a software system. Even though we do not use UML or UMLsec, it is worthwhile to mention that the concepts we use in our research could very well be included as part of UMLsec because we are adding code-level restrictions on the interaction between processes. Notions such as stability, discussed later on, could become part of the tagging proposed by Jürjens et al because they impose a condition that has to be respected by both participants during a concurrent interleaving.

Our research approaches security at the level of code by judging on interaction between different parts of code that become interleaved once there is some concurrent interaction between processes. On a larger scale, due to the interaction between different processes or threads, we observe the program as a whole where each individual part contributes a part to the overall outcome of a program. We notice that the more participants we have, that contribute to the overall outcome of a program, the more likely it becomes that the program opens up new avenues for attacks. On the higher level, we borrow some aspects from UML that allows us to describe the interaction between processes and threads.

Although UML has been used to model higher-level interactions at the level of users and systems, the markup employed can be applied to the lower levels of programming languages. For example, if we refer to signals in system programming, we can observe that the signal system has a client-server model where the process sending the signal can be seen as a client and the process receiving the signal can be seen as a server. The only restriction that permissions bring to the client-server model, is that a client is not able to send a signal unless the server process runs under the same user-id as the client. This restriction is the only condition for not being able to send a signal however, in case the daemon user account has been compromised, any signal can be sent to the daemon under the same user ID.

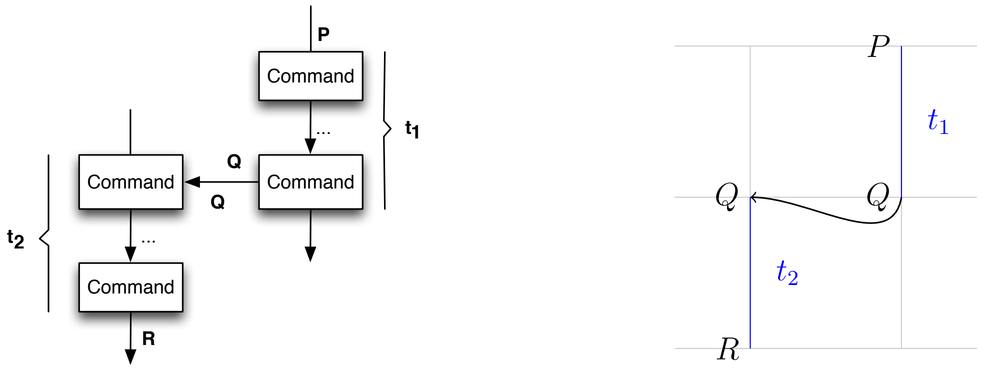


**Figure 2.9:** A side-by-side comparison of an UML diagram on the left hand side and a swimlane diagram on the right hand side. Control flow can be observed as a series of commands, snapped together by matching pre- and postconditions. This is similar to a typical UML diagram with parallel processes.

When applied to security and programs that are highly concurrent, we use swimlane diagrams (Figure 2.9) that are able to further refine the higher-level model of process interaction. While the higher-level UML interaction models deal with permissions and users, we apply the same concepts on the lower code-levels where each participant is modeled by using a client-server and attacker model. In that sense, for every interaction between two processes or threads, we offer a specification to which both the client and the server will have to comply in order to make sure that the interaction is secure. We cannot force the legitimate participants to comply with a specification we provide but we can make sure that if they conform to the specification, then we can say that the interaction is secure.

## 2.8 Conclusions

Tools such as Valut, Cylone, CCured, Split, CQUAL and the newer features of the GNU C Compiler provide the necessary language constructs to reason about software vulnerabilities, one can observe that they are limited to single processes. These dialects offer additional annotations in order to address the most common security flaws, such as buffer overflows or type violations. However, they do not address any kind of concurrent interaction between two different processes. These annotation-based language extensions have mostly been outdated. For example, Vault has been made obsolete and replaced by C# by Microsoft. During this transition, very few notions developed in Vault that address security, have been included in C# - for example, C# does implement regions and getters and setters but they do not address software security but rather code safety and optimizations.

On the other hand, abstract languages such as UML and UMLsec, described in Chapter 2.7 do provide a way of describing the concurrent interaction of processes. For the scope of this thesis, many concepts from UML and UMLsec, particularly the methodology of describing the interaction of concurrent processes has been leveraged and used in the "Taxonomy on Layers of Abstraction" Chapter 7.8 in order to reason about attacks and vulnerabilities.

CHAPTER 3

# FLAVORS OF LOGIC

In order to reason about attacks, several flavors of logic have been reviewed which could provide an insight on how to seal vulnerabilities and reason about code safety. From this chapter we leverage concepts and use them later for building swimlanes and to reason about vulnerabilities.

## 3.1 Propositional Logic

Propositional logic, also known as sentential logic or statement logic, is the branch of logic that studies ways of joining or modifying entire propositions, statements or sentences to form other propositions, statements or sentences as well as logical relationships and properties that are derived from these methods of combining or altering statements. The simplest statements are considered atomic units and hence propositional logic does not deal with the logical properties and relations that depend upon parts of statements which are not themselves statements on their own such as the subject and predicate of a statement. The most thoroughly researched branch of propositional logic is the classical truth-functional propositional logic which studies logical operators and connectives that produce complex statements depending on the truth values of the simpler statements used to build them up. This makes propositional logic to be a formal system for performing and studying logical reasoning and as such it has a precisely defined grammar.

Propositional calculus is a formal system in which formulas representing propositions can be formed by combining atomic propositions using logical connectives and a system of formal proof rules [56]. These rules allow us to derive other formula based on a set of formula that are assumed to be true. In turn, these would allow us to build a proof for a certain assumption by either pushing the axioms forward using the rules or symmetrically by tracing backward.

## 3.2 Linear Logic

Linear logic [37] is a type of substructural logic that denies the rules of weakening and contradiction. The concept of "hypotheses as resources" is the base of linear logic and it basically means that each hypothesis is consumed exactly once in a proof. Once a hypothesis is consumed, it can not be used again. This differs from the previous propositional logic we had where the judgment is based on truth, which may be used as many times as necessary. Linear logic becomes very interesting due to this consumption feature that has inspired an interesting solution to Zalewski's signal handler attack. It is clear that if a signal handler can be used only once then one should use some mechanism to dispose of that signal handler once it has been used. Similarly, by using the bang operator "!" one can illustrate both behaviors where a certain resource can be used only once or multiple times. This is very similar to the way that signal handlers are used in various implementations: as a consumable function that is used only once by unbinding from a signal identifier or several times where the signal handler is automatically rebound to a signal identifier.

Linear logic introduces a few new logical connectives. The most important being the linear implication $A \multimap B$ which says that an action $A$ is a cause of an action $B$. A formula $A$ can be regarded as a resource which is consumed by the linear implication. It is the most important feature of linear logic and also of the programming based on it. In classical logic the truth value of the formula A after the implication $A \Rightarrow B$ remains that same meaning that the same rule can be applied over and over again as many times as it is necessary. Classical implication can be ported using a modal operator and by writing $!A \multimap B$ meaning that we can use resource $A$ repeatedly. Linear logic also defines the "multiplicative conjunction" represented by $A \otimes B$ and saying that both actions $A$ and $B$ will be done. There is also the additive conjunction represented by $A \& B$ expressing that only one of the actions $A$ or $B$ will be performed at choice. Another one is the "additive disjunction" represented by $A \oplus B$ which also says that only one of the actions A or B will be performed yet it is unknown which one of them. Finally, there's "multiplicative disjunction" $A \parr B$ which is saying that if $A$ is not performed then $B$ is done, or vice versa: if $B$ is not performed then A is performed [54].

The bang operator "!" allows us to hardwire the classical logic version of $A \multimap B$. Writing $!A \multimap B$ (which is written in this form for the sake of example) is basically saying that $A$ would be consumed but we have an endless supply of $A$'s [66]. We also use a new rule called dereliction in order to select on $A$ out of our supply of $A$'s. Basically this is saying that if we have an endless supply of $A$'s, we select one out of them. The problem is that we can not apply $\multimap B$ to an $!A$; we can only apply it to one single $A$ and hence the need for dereliction. The axioms are also modified to reflect the change brought by the resource consumption concept.

Computer science relies on computational mechanisms such as function application, exception handling, method invocations, variable assignments and similar. It is thus necessary to make the mechanisms of these processes explicit by using a formal language. An event such as $A \Rightarrow B$ generally describes a transformation from $A$ into $B$. This is also extensively applicable to automated processes, more precisely Petri nets, where a clear protocol must be implemented by following a series of transformations. One key application of this resource management aspect of linear logic was the development of a functional programming language which replaces garbage collection by explicit duplication [51] operations. Other applications include analyzing the control structure of programs, generalized logic programming and natural language processing. One interesting feature is the ability to give a natural aspect of polynomial time computations in a bounded version of linear logic. This is done, again, by limiting the reuse of specified bounds.

Linear logic has been the inspiration for the work done on signal handlers in this thesis. The idea of consuming a resource which is present in Vault and handled by linear logic, was the starting point for building the non-persistent one-shot signal handlers. If resources such as keys or regions can be used in a linear sense so that they get removed from the address space, then the same reasoning could be applied to functions and more precisely to signal handlers. By making a signal handler one-shot, one would not need to worry whether the signal handler is re-entrant safe and it would definitely fix the problems encountered in Zalewski's study of signal handlers. Furthermore, later on, while going through the kernel-level implementation of signal handlers, we have seen that signal handlers do indeed have a flag which may be toggled to make signal handlers consumable. However, this was part of an effort to make signals portable sine the default behavior varies from a platform to another.

### 3.2.1 Vault

Coming back to Vault, we have to note that Vault brings some personal contributions based on linear and intuitionistic logic. This is based on the concept that a certain object or variable in a programming language may have a specific lifetime and it is easier (for example, in cases where we must free up a resource) to reason about them if we combine the two logics together. DILL [24] comes as an answer to that and explores the compatibility and transitions from one logic to the other. As a repetition, in intuitionistic logic we write constructs such as $A \Rightarrow B$ which can be seen as a function application: "given an $A$ we can produce a B". This, of course, can be seen as a function which can be applied as many times as necessary, taking a parameter and producing a result. In linear logic, we write constructs such as $A \multimap B$ which can be seen as a one-way function application: "given an A we can produce exactly one $B$ by consuming the $A$".

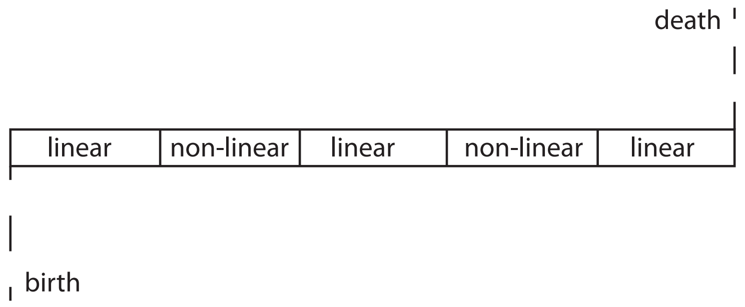


**Figure 3.1:** All objects are born and die with a linear type. However they may change types by using adopt and focus.

Such a function would resemble for example a `free()`, in which we take a parameter and, in this case, delete it after which that parameter can not be accessed again. By using linear logic one can efficiently describe such functions yet it is not sufficient and we need inuitionistic logic for other functions which do not consume parameters since it is obvious that by building a language solely on linear logic would produce unwanted results (every variable, function or object will need to be recreated every time and will be consumed every time after an operation). DILL [5] has sought to combine the two logics together by exploring how far one can combine inuitionistic logic with linear logic in order to acquire the functionality of both. This however leads to very complex constructs and ends up creating the need for another logic (which we describe in the following chapter). Vault solves the incompatibility between intuitionistic and linear logic by creating two new concepts: adoption and focus [29]. In Vault, all object die and are born as linear types. Non-linear objects are temporarily cast to linear by using the let! construct. The following figure is a representation of the lifespan of an object in Vault depicting the transition from linear to non-linear and vice versa through the means of adoption and focus. By using adoption and focus it is possible to change the type of an object however, as seen in the figure, all objects are born and die with a linear type as illustrated in Figure 3.1.

Vault uses adoption and focus for the transition from linear to non-linear and vice versa. Vault additionally implements tracked types and thus keys that protect linear types and guards non-linear types. If a key `k` is accessible then all normal operations are

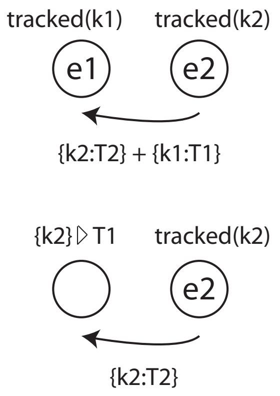


**Figure 3.2:** Adopting `e1` by `e2`.

allowed on the linear type. Similarly, if we hold all the guards for the non-linear type then all normal operations are allowed. The operations to convert from one type to the other are called adoption and focus. Since all objects are created as a linear type and since it is impractical to program without aliasing Vault uses adoption as a way to obtain aliases.

`adopt e1 by e2` takes an adoptee, say `o1` which is the result of `e1` and an adopter, say `o2` which is the result `e2` both of linear type and consumes the linear reference to `o1` by creating an internal reference from `o2` to `o1` (Figure 3.2). Thus, an adoptee has exactly one single adopter and the result of the adoption expression is a reference to a non-linear type to the adoptee. This adoptee's nonlinear type is then its previous linear type with only one top-level type constructor changed from linear to nonlinear. Access to `o1` is disallowed through the internal reference of the adopter and the non-linear reference to the adoptee is valid for the entire lifetime of the adopter. Then, any linear components of the object `o1` can not be directly accessed through the non-linear reference since that would lead to a shared access violation to objects of linear type. However, the linear components of `o1` may be accessed in the scope of a focus operation. When the adopter is disposed, all non-linear references to the adopted object become inaccessible. Adoptions stack so one can adopt several objects through multiple adoption expressions. Focus provides a way to temporarily view an object of non-linear type as a linear type. The turning point is that any type invariant of a non-linear object can be violated as long as no alias for the object can witness the violation.

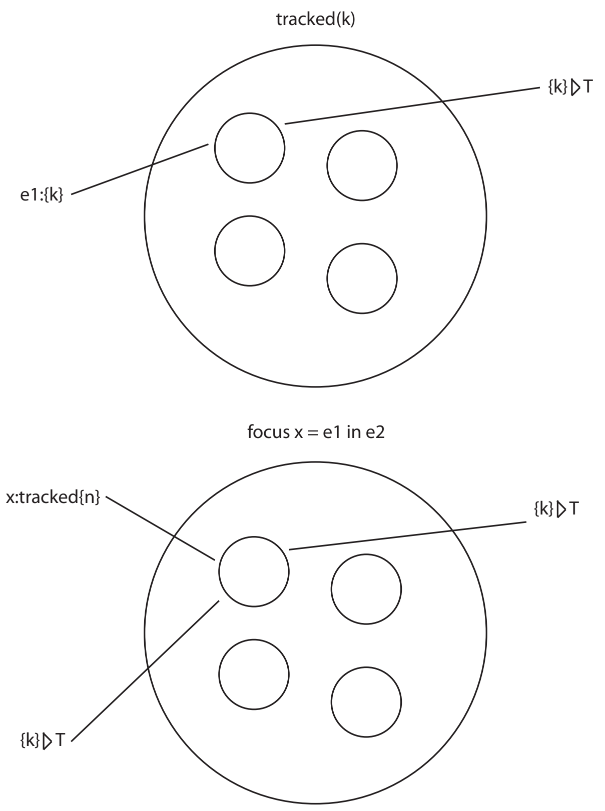


**Figure 3.3:** Temporarily viewing an object of non-linear type as a linear type by using focus.

The focus construct `x=focus e1 in e2` requires `e1` to evaluate to an object of a guarded type as illustrated in Figure 3.3. This object is called the focus object. The first thing to do is to bind `x` to the focused object and give `x` a tracked type `trackedn` for a newly created key. Now, because `x`'s tracked type, the expression `e2` can change the type associated with the key `n` and can thus access and replace linear components. Second, in order to ensure that no aliases can witness these changes, within the context of `e2`, the original guarding key is removed. In doing so one guarantees that no aliases of the focused object are accessible during `e2`. The focus is ended by revoking the temporary key `n`. The code example in Figure 3.4 illustrates how a pointer can be de-allocated using a non-reentrant function.

```

void resize({D} cell c) {
        focus x=c in
        free x.data;
        x.data = new array;
}

```

**Figure 3.4:** De-allocation of a pointer using a non-reentrant function `f`

## 3.3 Hoare Logic

Hoare logic [43] is a formal system invented by C. A. R Hoare and offers a way to reason about computer programs and control flow by formalizing the sequential execution of programs as a series of commands. Hoare does this by introducing the "Hoare triple" which, in its simplest form is structured as a precondition, a command and a postcondition. This concept can also be observed in automata, given that any action or command has an initial, and an end state, the transition from one state to the other is performed by an action. If that action or command does not terminate, the automata never commutes to the next state.

This lets us model transitions in programming because a program may be observed as a sequence of sequential steps [79]. Each of those steps makes a transition from a begin state by executing a certain command and then ending in an end state. By composing a sequence of commands we can illustrate the whole control-flow of a program.

## 3.4 Separation Logic

Separation logic [67] is a substructural logic and an extension of Hoare logic attributed to John C. Reynolds describing a way of reasoning about programs. Particularly, separation logic addresses several problems where other logics become hard to understand. Similar to the mix of intuitionistic and linear logic concepts, separation logic tries to solve certain issues pertaining to context splitting. Separation logic facilitates reasoning about programs that manipulate data structures [61], ownership transfers and modular reasoning between concurrent modules and brings the concept of local reasoning.

Most important system daemons make serious use of the heap (Apache and other system related daemons) but without a way to handle memory, heap verification is particularly difficult. With separation logic, we can address the problem by splitting the heap into several sub-parts and reason about them individually which is useful when dealing with concurrency.

## 3.5 Conclusions

We observe that the overview of the different flavors of logic could potentially provide the necessary basis for reasoning about vulnerabilities. For example, we can see that the attack performed by Zalewski in the "Research Context" Chapter 1.3, could perhaps be approached by making a linear usage of the signal handlers as we do in the "Taxonomy on Layers of Abstraction" Chapter 7.8. We can also see that similar ideas from linear logic have been used in Vault, so it remains to be seen what could be done for signal handlers. The question is whether the signal handling mechanism already has the ability to use the signal handlers linearly, or whether that would be an additional construct that would have to be implemented for the Zalewski attack.

CHAPTER 4

# SIGNAL AND EVENTS

Signals are supported by the library but not commonly used on platforms such as Windows because they are designed to operate in an environment that must provide the means to interrupt daemons. Windows platforms seem to be more oriented toward graphical user interfaces where system daemons are uncommon. Thus, Windows makes use of an event and callback system that is meant to asynchronously process messages sent by the environment. This works by using an event queue where each process can schedule an event so that it may be processed at a later time. The Linux alternative for event and callbacks meant to be used with graphical user interfaces, is an extension of the signaling mechanism that has been extended. One such example is the `libsigc++` [44] library primarily meant for the gnome graphical toolkit (GTK).

The `libsigc++` library addresses issues such as type safety, which is an inherited problem from the daemon-style signals. One problem that the `libsigc++` library addresses is that any type of data may be passed without any restriction imposed on the types of variables that are passed to the signal handler. When a callback is triggered, some data may be passed which is allowed to be of any type. If that data sent to the callback is of the wrong type it is possible that the expected type does not match the type of the data resulting in a type mismatch error. `libsigc++` addresses that problem by implementing "slots" which are able to hold a reference to the different callback types.

The process of attaching and detaching a signal handler from a signal identifier is brought over from the system daemon variant of signals. At any point during the execution of a program a signal handler may be attached or detached from a signal identifier. Another feature of the `libsigc++` library, is the ability to define custom signal types. These custom signals are presented as a template class which allows the programmer to define their own type of signal. This is a particularly interesting concept and comes as novelty for signals because in the system daemon variant the signal identifiers are a predetermined set which does not allow any extensions.

A rough illustration of the problem addressed by `libsigc++`, in contrast with C, may be seen in the example shown in Figure 4.1.

```
void click(void *ptr);

int main(void) {
        GtkWidget * button = gtk_button_new();
        char data[] = "Data passed to the click() callback.";

        register_click_handler(button, click, data);
}
```

**Figure 4.1:** Using sigc++ together with GTK interfaces in Linux systems.

The example code, generates a new GTK button and creates a character array that will be passed to the `click()` callback whenever a button is pressed. We can see that the `click()` function takes a `void` pointer as an input parameter allowing any data type to be passed. Since there are no restrictions imposed on the type of the data passed, the programming logic in the `click` callback is unable to infer how the passed data should be handled.

`libsigc++` solves that problem by enforcing type safety and by imposing restrictions on the type of data that may be passed. An illustration of this can be seen in Figure 4.2 is written in C++ and is using `libsigc++` .

```
class ButtonSignal {
        public:
        ButtonSignal();
        void run();
        sigc::signal<void, std::string> signal_detected;
};
void click_handler(std::string s) {
        cout << "Received: " << s << endl;
}
void main(void) {
        ButtonSignal bt;
        bt.signal_detected.connect(sigc::ptr_fun(click_handler));
        bt.run();
}
```

**Figure 4.2:** Using template parameters in the signal handler `click_handler` using sigc++.

The template parameters `void` and `std:string` enforce check for consistency at compile time. Also, one may observe that the type is enforced to `std::string` which makes this example type safe. Here, the `connect` method is similar to a signal binding however, using `libsigc++` one can choose to disconnect so that the handler will not run again if a new signal is received.

## 4.1 Signal Implementations

The following chapter provides a description of the current kernel 2.6 implementation of the signal mechanism based on "Understanding the Linux Kernel, Second Edition" [9] that refers to the old mechanism (before the Linux kernel version 2.4) of signal handling. We have gone by-hand through the most recent kernel and followed the reasoning of the code in order to provide an accurate description of the signal-handling mechanism as well as some insight on how the new emerging signal mechanism based on `sigaction` has evolved. Signals have been used for a long time and started off as a small number and have grown throughout the years to a full complement of around `60` individual signals, some user- and some kernel space signals [75, 55].

A signal can be either generated by the user (synchronous signals), either by sending the signal directly from a terminal, sending the signal from within another process, or they can be triggered by the kernel (asynchronous signals), for example, illegal instructions or I/O termination. Signals are executed in user mode and even if the program is running in kernel mode, the kernel will defer sending the signal until the process drops down to user mode. It is important to note that a process may only send a signal when it either has the `CAP_KILL` capability or when the effective user ID that the sending process is running under is the same as the target process [38]. The exception to that rule is the first process, on linux systems, process number `1`, the `init` process. This offers a certain level of protection, due to the fact that an attacker cannot arbitrarily send a signal to any process on a compromised system. When sending a signal, there are three possible outcomes:

- The signal is ignored, for example if the signal is not handled in the code.
- The signal is handled using a user-defined signal handler.
- Some default action for the specific signal is executed. For example, certain types of signals such as `SIGSTOP` or `SIGKILL` cannot be caught.

The kernel saves information about which signal and signal handler is to be executed in the process table. It simply toggles a flag on the specific signal in an array of all possible signals. Consecutive signals, or concurrent signals for that matter are ignored and aren't buffered up. More precisely, the first signal must be processed before a second signal of the same type can be flagged in the process table.

The following code execution illustrate the atomicity of the signal handlers showing how one signal after the other gets buffered. In this case, two subsequent `SIGINT` signals were delivered to the program running under the process identifier 22375 as illustrated in Figure 4.3.

```
void sig_one(int sig) {                PID: 22374
   static int passes=1;                Got SIGINT
   printf("Got SIGINT\n");             1 passes
   sleep(10);                          Got SIGINT
   printf("%d passes\n", passes);      2 passes
   passes++;
}
int main(void) {
   printf("PID: %d\n", getpid());
   signal(SIGINT, sig_one);
   sleep(20);
}
```

**Figure 4.3:** Running the signal handler `sig_one` twice in succession. The output is presented on the right-hand side.

One signal got processed, and during the `sleep(10)` of the signal handler, a second `SIGINT` was sent. The process waited for the first signal to get processed and then executed the signal handler again showing that the `SIGINT` was buffered.

However, if we repeat the experiment with two different signals, we notice that `sig_one`'s sleep cycle was interrupted by the second signal `SIGHUP`, `sig_two` was executed and then execution came back to `sig_one` which printed out `1 pass` as illustrated in Figure 4.4.

Using the newer interface `sigaction` one can also choose to clear this table or the process can be made to ignore a certain signal. Regarding concurrency, POSIX, in newer variants such as newer kernel distributions make a clear distinction between process signals using the normal `kill()` function and thread signals through `pthread_kill()` [25]. The question that arises is who actually receives a signal in case of a program spanning multiple threads. In case of user generated signals, or synchronous signals, are delivered to the thread which has not blocked the signal and asynchronous signals (those generated by the kernel from illegal instructions, for example, faulty I/O) are delivered to the thread which triggered the signal in the first place. In case of a symmetric multi-processing systems (SMP) where several threads are distributed amongst processor, the kernel manages to send a message to all CPUs involved in the execution of that certain process making sure

```c
void sig_two(int sig) {
   printf("Got SIGHUP\n");
}
void sig_one(int sig) {
   static int passes=1;
   printf("Got SIGINT\n");
   sleep(10);
   printf("%d passes\n", passes);
   passes++;
}
int main(void) {
   printf("PID: %d\n", getpid());
   signal(SIGINT, sig_one);
   signal(SIGHUP, sig_two);
   sleep(20);
}
```

```
PID: 22401
Got SIGINT
Got SIGHUP
1 passes
```

**Figure 4.4:** Delivering two signals `SIGINT`, respectively `SIGHUP` and using two different signal handlers `sig_one`, respectively `sig_two` to process the signals. The output of the test is illustrated on the right hand side.

that all threads receive the signal at the same time [73].

One thing to note is that signals can be regarded as the software equivalent of hardware interrupts. More precisely, once a signal has been delivered, execution would stop exactly at the point where the signal was delivered and switches over to the signal action. Either a handler is installed to process the signal or the process ignores it. When the handler returns, execution switches back to the point where the signal was initially received and the program resumes from that point. The sense of concurrency comes from the fact that a signal may be triggered at any point in time and the execution of the main event loop will be halted until the signal action has returned. However, the the signal handler does not run concurrently with the main event loop. In case two signals of the same type are delivered, the second signal will be discarded. This is not the case if the two signals are different but they will be stored in a buffer in the process table waiting execution.

## 4.2 Sending a Signal

Whenever a process or kernel task wishes to send a signal to a process, the kernel generates the signal for that process [9]. In order to do that, the kernel relies on several functions, the most notable of those being the following:

1. `do_send_sig_info()` - This is the standard way of generating a signal under the current kernel (as of 2.6.35).

2. `force_sig_info()` - This function forces a process to react to a signal, even by overriding `SIG_IGN` and setting it to `SIG_DFL`, if necessary.

There are other available functions which are variations of previous, as well as several functions dedicated to broadcasting signals to process groups. `do_send_sig_info()` is a wrapper which first checks whether the requested signal identifier is within valid bounds as illustrated in Figure 4.5.

```

if(valid_signal(sig))
    return -EINVAL

```

**Figure 4.5:** Checking whether a requested signal identifier is within valid bounds.

If it is valid, it acquires the signal spinlock of the receiver process, send the signal and then release the spinlock. This is done via the functions `lock_task_sighand`, respectively `unlock_task_sighand()` and can be observed in the kernel source as illustrated in Figure 4.6.

```

if(lock_task_sighand(p, &flags)) {
    ret = send_signal(sig, info, p, group)
    unlock_task_sighand(p, &flags);
}

```

**Figure 4.6:** Acquiring a spinlock in order to be able to send a signal.

Acquiring the spinlock is done within the `lock_task_sighand()` function by polling the RCU. If the spinlock is acquired, the `send_signal()` function does a check for process namespaces, a new feature of the 2.6 kernels and passes the structures to the `__send_signal()` function. The `__send_signal()` function asserts the acquired spinlock and then proceeds to fill in the `siginfo_t` table with a new signal item.

```

if (q) {
    list_add_tail(&q->list, &pending->list);
    switch ((unsigned long) info) {
        case (unsigned long) SEND_SIG_NOINFO:
        /* ... */
        case (unsigned long) SEND_SIG_PRIV:
        /* ... */
        default:
            copy_siginfo(&q->info, info);
            if (from_ance?tor_ns)
                q->info.si_pid = 0;
                    break;
    }
}

```

**Figure 4.7:** Creating a new signal by filling the `siginfo_t` table with the new signal item.

This varies from the previous kernels in that the case distinction here is made between the defined value of `SEND_SIG_NOINFO` identifying the signal as being sent by an userland process, or `SEND_SIG_PRIV` for a signal sent by a kernel function. Additionally process namespace handling is done in the default switch block as seen in Figure 4.7.

Another new notable feature here is that, if the signal cannot be added to the queue since it would exceed the maximum number of queued signals or if there isn't any free memory left, the kill() must not be allowed to fail since it is the only way the user may terminate the process and recover the system. If the signal generated was a real time signal sent by the user using anything other than kill(), the kernel aborts the sending by returning EAGAIN as illustrated in Figure 4.8.

```

trace_signal_overflow_fail(sig, group, info);
return -EAGAIN;

```

**Figure 4.8:** If the signal queue is full or no free memory is available and if the generated signal is a real-time signal then the kernel aborts by returning EAGAIN.

Otherwise, the signal is sent but without the info structure to ensure that the process receives the signal as seen in Figure 4.9.

```

trace_signal_lose_info(sig, group, info);

```

**Figure 4.9:** Sending the signal without the info structure in order to make sure that the signal is received, even if there is no free space left.

The signal is then added to the queue for delivery (Figure 4.10).

```

        signalfd_notify(t, sig);
        sigaddset(&pending->signal, sig);
        complete_signal(sig, t, group);
        return 0;

```

**Figure 4.10:** Adding a signal to the queue and scheduling a delivery.

If the __send_signal() function succeeds and the signal is not blocked, the kernel tries to wake up the destination process by setting the sigpending flag, sending an inter-processor interrupt in case the signal is currently running on a different CPU and then proceeds to check whether the process is interruptible so it will be able to process the signal. Finally, the unlock_task_sighand() releases the spinlock and returns the error value of __send_signal().

The second function we mentioned, `force_sig_info()` is the same with the only difference that it resets the signal to the default state `SIG_DFL` in case the destination process had previously blocked the signal via `SIG_IGN` making sure that the signal will go through.

## 4.3 Receiving a Signal

The kernel signal handling works on a simple queue principle. It spins around and dequeues pending signals while other signals are enqueued. This is done by the function `get_signal_to_deliver()` which repeatedly invokes `dequeue_signal()` until no non-blocking pending signals are left as illustrated in Figure 4.11.

```

for(;;} {
    ...
    signr = dequeue_signal(current, &current->blocked, info);

    if (!signr)
        break; /* will return 0 */

    if (signr != SIGKILL) {
        signr = ptrace_signal(signr, info, regs, cookie);
        if (!signr)
            continue;
    }

    ka = &sighand->action[signr-1];
    ...
}

```

**Figure 4.11:** The kernel then proceeds to dequeue the signals in the queue until no signals are left.

The first check is whether there still is a pending signal on the queue. This is done by checking whether `dequeue_signal()` returns a signal or not. At the end of the loop, the `ka` variable gets the address of the action data structure associated with the signal to be handled.

Depending on the contents of the signal data structure, a process to which a signal was delivered reacts in one of the following possible ways:

1. Ignores the signal explicitly by having previously set the flag SIG_IGN on the current signal mask. When processing the signal, a simple check is made to determine whether the SIG_IGN flag is set as illustrated in Figure 4.12.

```

if(ka->sa.sa_handler == SIG_IGN) {
    ...
    continue;
}

```

**Figure 4.12:** If the signal handler has been decoupled from the program, then the default behavior is to ignore the signal. This is done by checking whether the handler is set to SIG_IGN.

2. Catches the signal by previously binding to the signal with a signal handler. Once the signal is received, the kernel changes the execution state of the process and the signal handler associated with the signal runs. this corresponds to a non-standard code jump in which the process state is stored and control flow switches to the signal handler attached to the signal. After execution of the signal handler, the state is restored and the process resumes from where it left off. If a signal handler has been bound, the kernel invokes handle_signal.

3. If the signal is neither ignored explicitly nor trapped by a signal handler, the process executes the default action associated with the signal. There are 31 traditional signals and an extension of 32 modern real-time signals, each corresponding to a default action which will be triggered unless the signal is specifically ignored or caught. As an exception to that rule, there are two signals which cannot be ignored or caught by the process: SIGKILL which terminates the process immediately and SIGSTOP which suspends the execution of the process. The difference between SIGKILL and SIGSTOP is that SIGKILL terminates the process immediately whereas SIGSTOP suspends the process allowing continuation by the delivery of another signal SIGCONT.

The checking for the signal is done by switching on the signal number. For SIGCONT, SIGCHLD, SIGWINCH where the default action is to ignore, the kernel just passes through using continue as illustrated in Figure 4.13.

```

case SIGCONT:
case SIGCHLD:
case SIGWINCH:
    continue;

```

**Figure 4.13:** For signals where the default action is to ignore, the kernel puts them together and passes though to the next signal without processing them.

## 4.4 Handling Signals

Handling signals varies between platforms because it involves switching back and forth between userland and kernel space. We provide a general overview of the signaling mechanism, sometimes lightly based on the x86 platform when a concrete example is needed.

As we mentioned before about the possible signal sending outcome, if a handler has been installed by a process receiving the signal, the kernel enforces the execution of the handler by running handle_signal. Once the signal is handled by the process, the kernel makes sure that the other signals are processed on the next invocation of do_signal. This is done by returning from the handle_signal function immediately after the handle_signal call as seen in Figure 4.14.

```

if (handle_signal(signr, &info, &ka, oldset, regs) == 0) {
    current_thread_info()->status &= ~TS_RESTORE_SIGMASK;
}
return;

```

**Figure 4.14:** The process receiving a signal identifier is instructed to handle that signal identifier immediately. The kernel enforces that behavior with the handle_signal function.

Signal handlers have to run in user mode, however handle_signal runs in kernel mode. When a signal is received, the process switches into kernel mode. Upon return `do_signal` calls `handle_signal` which sets up the user mode stack. This is done via the `__setup_frame` and `__setup_rt_frame` calls depending on whether the signal was a real time signal or not.

When the process switches back to user mode, the installed signal handler's address has already been pushed onto the user mode stack and the program counter adjusted to point to the signal handler. This is the mechanism through which the kernel forces the execution of the signal handler in user mode.

When the signal handler returns the return block placed by the kernel on the user mode stack by the previous `__setup_frame` and `__setup_rt_frame` calls is executed which invokes the `sigreturn` call responsible for copying back the contexts. The original context of the program is also restored via the `restore_sigcontext` function call and the program may resume execution.

After the user mode stack is set up, `handle_signal` goes through the flags associated with the signal. This is where, for example, the decision is made between a persistent rebinding to the signal or a one-shot signal handler type as shown in Figure 4.15.

```

if (ka->sa.sa_flags & SA_ONESHOT)
     ka->sa.sa_handler = SIG_DFL;

```

**Figure 4.15:** At this point the type of the signal handler is checked. If the signal handler is meant to rebind to the signal identifier and has the `SA_ONESHOT` flag toggled, then the default behavior is restored.

By checking the `SA_ONESHOT` flag, the signal is reset by forcing the behavior of the signals back to the default behavior flag `SIG_DFL`.

## 4.5 Signal Handling and Exceptions

We reason about the the current signal handling mechanism in order to provide a better and more secure way for dealing with signal interleaving. We aim to prevent the attacks such as the ones described in Zalewski's paper [89], the WU-FTP daemon exploit [39] or the Sendmail exploit by redesigning the signal handling mechanism and introducing exceptions as substitutes for the signal handler functions. On an abstract level, we use continuations and traces to reason about the exceptions handling mechanism. On the implementation layer, an extension is possible using the already existing exception mechanism, perhaps one can enable signals to be trapped by using regular `try...catch` blocks. By extending exceptions with signals, we could also trigger exceptions from outside of the process by delivery a signal. We have cases, such as Sendmail, where a `longmp()` is used together with a signal handler in order to emulate an exception behavior and break out of the main event loop. Using the current exception mechanisms, there is no way to trigger exceptions within another program. We have different types of IPC to allow message passing between processes. However these are heavyweight systems with intricate methods of operation and we believe that the lightweight possibility of triggering exceptions in a running process externally would greatly increase the functionality and flexibility of a program without overloading it with additional data-passing. As a general guideline, a signal handler is meant to run for a limited amount of time and do as little as possible. Any signal handler which exceeds a certain level of complexity should probably be replaced by some other form of IPC which is more suitable for a concurrent context.

## 4.6 Persistent and Non-persistent Signal Handlers

One interesting property of signals is that they may be declared to be one-shot, thereby granting a signal handler the ability to be consumable, so that they can only be called once. The motivation for this distinction is the exploit described in Zalewski where a signal handler containing non-reentrant function is called twice by delivery of two carefully timed signals and thereby causing a double deallocation memory corruption.

The idea of reasoning about interchangeable persistent and non-persistent signal handlers is inspired by the two different implementations of the signal mechanism. On Windows systems, a signal handler is non-persistent by default, meaning that after the signal has been processed, the process will not re-attach to the signal and a second delivery of that signal will be ignored. On POSIX systems, the signal handler is re-attached to the signal after it has been called, thus making the signal handler persistent and allowing it to be called several times.

Because of implementation differences, we think that it would be useful to be able to dynamically switch between the two behaviors. By doing so, one could use a non-persistent signal handler for the exploit presented in the Zalewski paper. Intuitively, since the problem described in that paper is a non re-entrant signal handler, if do not allow the signal handler to be called twice, then one pass through the signal handler would free up the referenced memory and then decouple the process from the signal identifier. Thus, a second pass through the same signal handler would not be allowed which prevents the double deallocation memory corruption.

There are several cases, where the distinction between persistent and non-persistent signal handlers can be seen as a feature rather than an error of portability as described by CERN. The ability to choose between the two types of behavior would fix a wide range of exploits that we have gathered during the literature overview.

Reasoning using control-flow we can find three types of signal handler mechanisms with different behaviors that can be found in various implementation. We illustrate the main body `b` and the handler `h` using separate lanes, similar to an UML diagram where each process is on a separate lane. Even though signal handlers do not run concurrently with the main thread, signals may interrupt a main thread at any time and then execute some code.

In Figure 4.16, we illustrate a typical scenario that can be found on Windows platforms that have a signal implementation. In this case, the main thread `b` is interrupted at some point by the signal handler `h` that runs some code and then returns precisely where the main thread was interrupted. As we can see, the pre-condition for the signal handler `h` is marked with $P_h$ whereas the post-condition of the signal-handler is `emp` indicating that

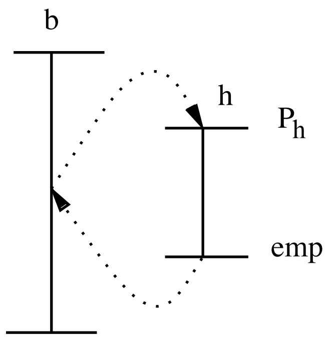


**Figure 4.16:** Non-persistant signal handler: Control flow transfer from the main thread `b` to a signal handler `h`.

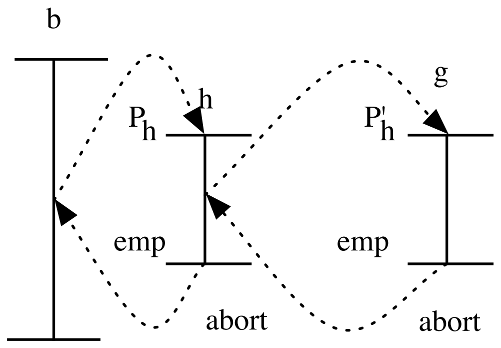


**Figure 4.17:** Aborting signal handler: The two handlers `h` and `g` can interrupt each other before returning to the main thread `b`.

the signal handler will not run again. Later on, using rely and guarantee we are able to say more about the signal handler and the pre- and postconditions but at a glance we cannot infer too much about what the signal handler's post-condition will be. Implementation-wise, the signal handler `h` will not be re-installed and will run only once - a programmer would have to re-install the signal handler again after the first run in order to allow the signal handler to run multiple times.

Comparable to exceptions, two installed signal handlers `h` and `g` can interrupt each-other (Figure 4.17) if a second signal is delivered while the first signal handler runs. Signals and signal-handlers can be seen as a rudimentary form that proceeded exceptions due to the fact that we are able to nest signal handlers using different signal identifiers.

In the last scenario in Figure 4.18 we have the UNIX-style signal handlers that automatically re-attach themselves after being run. In this case we have a pre-condition $P_h$ and a post-condition $Q_h \vee emp$ the outcome being either some post-condition from the

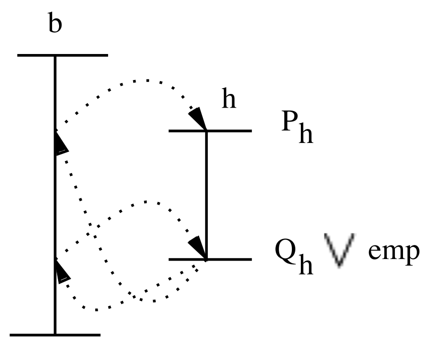


**Figure 4.18:** Aborting signal handler: The two handlers `h` and `g` can interrupt each other before returning to the main thread `b`.

signal handler `h` or `emp`.

In all cases we can observe that even though the signal handler is not a self-standing thread that runs concurrently, the signal handler can interrupt the main thread at any time which would change or invalidate the original precondition for the body. This is of particular interest to security because at any point in time for the duration of the program, a signal handler may interrupt the program and alter some shared data-structures and when the signal handler returns, the main body may not expect the shared data to have changed. Unless the programmer is ready to install and uninstall signal handlers, the program has to be able to adapt to any changes to shared data structures. Since most operations are not atomic, just running the signal handler at an inappropriate time may corrupt some part of memory - most typical cases include interrupting the body while some file handling operations are running.

This leads to a whole avenue of attacks where vulnerabilities arise due to control flow transfer between different parts of code at inappropriate times and are remarkably easy to commit in a highly concurrent environment.

## 4.7 Conclusions

Unfortunately we have seen in Chapter 4.1 that signals such as `SIGSTOP` and `SIGKILL` cannot be caught and handle. This may have security repercussions because if an attacker would have the necessary permissions to send the signal then the daemon would suspend or terminate, respectively.

However, from the overview of signals and event-handling mechanisms, we have obtained the necessary information to be able to seal vulnerabilities such as the Zalewski attack. The signal handling mechanism benefits from the one-shot flag that could make a signal handler decouple and run only once which is explained in the “Handling Signals” Chapter 4.4. This allows us to distinguish between persistent and non-persistent signal handlers, explained in the “Persistent and Non-persistent Signal Handlers” Chapter 4.6 which is precisely what is needed to reason about the signals or event-based vulnerabilities as well as potentially offering solutions.

It is important to note from the “Persistent and Non-persistent Signal Handlers” Chapter 4.6 that signal handlers do not run concurrently with the main thread. This simplifies the analysis of signal-based attacks because commands would be then composed sequentially rather than in parallel. Nevertheless, we can see from the “Sending a Signal Chapter” 4.2 that a process can send a signal at any moment, and that the receiving process has to be ready to handle that signal. This means that while within the receiving process the commands are composed sequentially, a signal can arrive at any time so that sending and receiving signals occurs concurrently between processes. sdsdf

# CHAPTER 5

# TAXONOMY ON LAYERS OF ABSTRACTION

From the gathered literature overview, we have seen that "Time and State" vulnerabilities have not been studied in detail, particularly the attacks related to "Signals and Events" and "TOCTTOU" vulnerabilities. The purpose of this thesis is to bring order to only the vulnerabilities related to "Time and State" and it seems possible that the taxonomy can be extended to other types of vulnerabilities. The taxonomy presented here is thus an analysis of "Time and State" vulnerabilities, with the purpose of classifying them, that focuses primarily on "Signals and Events" and "TOCTTOU".

One of the reasons that prompted this research was that we noticed that "Signals and Events" and "TOCTTOU" are not self-standing isolated classes of vulnerabilities and that they may have connections to other vulnerabilities. As an example, we take an additional step to show that "DoS" is also part of "Time and State" and that "DoS" can be observed as having connections to both "Signals and Events" and "TOCTTOU".

When we refer to attacks, we refer to the methodology that is used to exploit a vulnerability in a software package. However, the taxonomy classifies vulnerabilities rather than attack patterns. Different tools (such as traces, RG, stability, etc...) are used to reason about why the attacks are happening, what vulnerabilities a software package has and further, to illustrate the possibilities for preventing those attacks.

The two terms, "attacks" and "vulnerabilities" are not interchangeable, for example, we may study an attack-pattern that exploits a "TOCTTOU" vulnerability but the vulnerability and the means to prevent it is classified in our taxonomy, rather than the attack pattern. We may analyze the methodology used to exploit a vulnerability, but the taxonomy offers defenses against those attacks.

Where language constructs from the “Secure Languages and Tools”, Chapter 2 address software security, they fall short in addressing concurrency issues. However, UML and UMLsec offers a framework to describe this interaction and based on UML, and the concepts from the “Flavor of Logic” Chapter 3, this chapter shows how one could reason about attacks and vulnerabilities.

Vulnerabilities in software packages and attacks that exploit them can take many different forms. There have been several attempts of bringing some order to classifications, variously called “Top Ten Vulnerabilities”, “Seven Deadly Sins” or “Pernicious Kingdoms” [78]. The latter classification by McGraw et. al is perhaps the most scientific one because it tries to classify vulnerabilities by borrowing the idea of a taxonomy from biology. In doing so, vulnerabilities can then be classified using a hierarchy which shows that vulnerabilities are not unrelated but that they can inherit certain traits. One such kingdom, for instance, is malicious input that encompasses a wide-range of vulnerabilities. As standard techniques of input sanitizing are more widely adopted, McGraw predicts that more sophisticated attacks will become increasingly dangerous, such as the class of “Time and State” attacks.

The taxonomy, as an extension to a workshop paper [16] and elaborated in the “International Journal of Secure Software Engineering” [17], tries to reason about attacks and attempts to build a taxonomy by studying attack patterns from different perspectives. We call these perspectives, ”layers of abstraction” because vulnerabilities can be studied on different levels: starting from high-level concepts (such as traces, rely-and-guarantee, stability), going through a level of fixes that could be applied in batch, and down to code-level fixes that can be applied to individual software packages. For example, a vulnerability that can be found in a software package may be reasoned about using logic, then a general fix could be applied to the whole codbase or that particular vulnerability

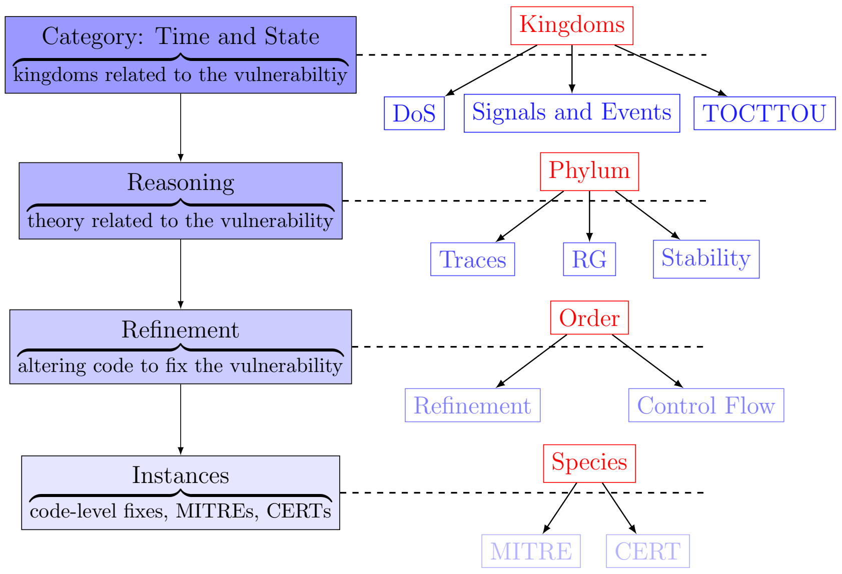


**Figure 5.1:** The taxonomy is annotated using McGraw's et al. terminology. Each layer describes a level of abstraction and every vulnerability can be classified by following the tree structure of a given attack. The upper layers are populated with abstract concepts such as "TOCTTOU", "Signals" and even more broadly"DoS" and reach down to lower layers where attacks distinguish themselves by local defects in a software package.

could be fixed as a single instance.

For example, a software engineer may fix a vulnerability on the code layer based on the attack pattern, however if that vulnerability is part of an important class of attacks, for example the "TOCTTOU" class of attacks, then that should be an indication that there may be other flaws within the program that may have to be addressed as well. The previously mentioned classifications do not hint to that possibility, but rather list vulnerabilities by citing the vulnerable code from the software package and summarily list the reasons why the flaw exists.

Similarly, the fixes that the other taxonomies present are local and pertain to a certain code-segment and they do not illustrate a general methodology that could be applied for the whole code-base. While companies such as MITRE and CERT list vulnerabilities separately (and exhaustively), the "Taxonomy on Layers of Abstraction" can group these disparate mentions of vulnerabilities and classify them together in a tree-like structure.

The terminology of "Kingdoms", "Phylum", "Order" and "Species" are only crudely related to our taxonomy and we adopt only the structure of the biological taxonomy. We use that terminology in order to provide a distinction between abstract security concepts and code-level safety issues on the lower layers of the tree.

Compared to biology, in terms of security, vulnerabilities with similar traits on the upper layers will be grouped together. We limit the article to the kingdom of Time-of-Check-To-Time-of-Use ("TOCTTOU") and "Signals and Events" as illustrated in Figure 5.1.

In biology, the "Kingdom" [72] rank is reserved for very high level classifications with a broad variety of descendants. McGraw's taxonomy, as well as our taxonomy follows the same principle. The very upper layers are reserved for abstract concepts which propagate to the lower levels of the taxonomy tree. We limit the article to the kingdom of "TOCTTOU" and "Signals and Events" which are both part of the "Time and State" category.

The "Phylum" in biology is a grouping of organisms based on a general abstraction of structure [81]. In our taxonomy, the "Phylum" is populated by the abstraction layer holding general concepts related to the program structure such as traces, states and predicates.

The connection between our taxonomy and the rank "Order" is that biological ranks group elements together based on small but important differences. For example, Zoology makes a distinction between moths and butterflies - which is not a trivial distinction. In our taxonomy, "Order" represents general elements of flow control and refactoring [32] the result of which may slightly change the code but sufficient enough in order to distinguish between a program and the re-factored equivalent.

"Species", being the last layer of the biological classification, represents individual instances of the upper ranks. Similar to biology, where around 10 million different species of bacteria are distinguished, our taxonomy reserves this layer for all known and classified instances of a certain vulnerability. For example, in various software packages and listed on the numerous software vulnerability sites such as CERT or CVE.

Although these ranks were selectively chosen with the intent of relating to McGraw's classification, it is plausible to extend the number of layers. For example,"Time and State" would be a superior layer that could be added before the rank of "Kingdoms", perhaps named "Domain". For example, we explore software vulnerabilities stemming from the code-level however, privacy and cryptography might be other high-level concepts, perhaps placed at the level of "Phylum". One instance thereof, might be information leakage or weak cryptographic keys and would extend the taxonomical tree.

DoS itself is an atypical member of the "Time and State" kingdom because it is frequently observed by effect rather than cause. We can observe a distinction between two types of DoS attacks which we give as examples in order to elucidate why DoS appears independently in taxonomy trees that do not seem directly related to DoS itself.

When Zalewski attacks Sendmail, or when shaun2k2 attacks Wu-FTPd, the purpose is to gain access to the system but, as a consequence, those services are disrupted. For a high-traffic service, that results in a DoS for the users currently connected and relying on that service. This type of attacks seem to force the program into an a state that it was not designed to handle.

One can recall hardware DoS attacks that used a light cast on a processor in order to flip bits [46]. From a technical perspective, we can see hardware attacks as a form of DoS because the processor is flooded in order to make the currently running program transition to some favorable state for the attacker. This is another example that falls within the Zalewski and shaun2k2 type of DoS because it plays on diverging control flow within the program even though the means of exploitation does not occur at the software level but rather at the hardware level. This is also closely related to the confused deputy problem mentioned in Chapter 5.2.1 because the software layer is not aware of the pressure exerted by the hardware layer that is able to manipulate control flow in the program to be exploited.

On the other hand, we have direct, software-based DoS attacks that are based on resource exhaustion. Network floods, shell bombs are some examples as well as DoS attacks based on regular expressions. The first two are perhaps the most trivial examples because it is obvious that their sole purpose is to consume resources. The shell bomb really just forks itself recursively, claiming more and more resources from the operating system and network floods make requests that a server is inclined to satisfy. We have found that the resource exhaustion based DoS seems to be a more predominant attack pattern but that the flow control based DoS is a clear indication that some service is being exploited for the sake of gaining access. Specifically, some attacks may result in a DoS, although the intent of the attack was to gain access. DoS, as we will notice further on, shows up many times, implicitly, as a consequence, rather than being an intended part of the attack.

With the ever increasing threat of botnet-type [1] attacks, it is important to see resource exhaustion DoS as one of the most savage attacks but it is equally important to mention that such attacks also impose pressure on the attacker's resources and that they cannot be maintained indefinitely. As such, resource exhaustion-based DoS is a primitive form of attack that, from the attacker's perspective, has a temporal benefit and is far distant from the eloquence of a carefully staged attack.

## 5.1 Overview

We leverage some technical concepts from programming language theory such as predicates, traces and program refinement as well as depicting traces by using "swimlane" diagrams. Given these tools, we then reason about TOCTTOU in Section 6 and signal handling attacks in Section 7.

The "Background" section 5.2 summarizes all the formalities that are used later on in the article. Although we use pictographic representations of traces, we follow-up with a section that instantiates the formalities to practical scenarios, followed up by case-examples of vulnerabilities in some of the most popular software package. For every example, we offer the reader some hints on refinement and how issues could be fixed by moving, removing or encasing misbehaving parts of code.

As a result, a wide range of software packages, including major system daemons have been analyzed using the proposed taxonomy on layers of abstraction. The results are summarized in the figures in the “TOCTTOU” Section 6.9, “Signals and Events” Section 7.7. We additionally offer a table in the “Taxonomy” Section 7.8 which maps diagrams to software.

In the end, we conclude that similar attack patterns can be classified together if they share the same tree-based representation.

## 5.2 Background

In order to reason about program flow, we make use of traces [23] which represent a series of state transitions allowing us to reason about control flow. Concurrent interaction is difficult to track because it occurs at runtime and because there may be multiple points of failure in a program that may lead to an opportunity for an attack. Also, it is worthwhile to note that most of the time it is not clear what parties or threads are involved while the program runs. A program is a series of sequential steps and, with very few exceptions, these steps are not executed atomically. Every step executes some command or performs some action which may establish a small invariant for the next step to be executed. As an example, these invariants can range from assuming that files exist, or that a certain flag has been set to some value, or that some concurrently running thread has finished executing and other similar examples. These invariants can become easy to violate in cases where users have the ability to influence a program - either directly, by accessing some exposed part of the API (for example, system daemons that offer an interface to trigger actions within a program) or indirectly by modifying something in the environment where the program runs. Once invariants are violated, the program may make false assumptions about what is going on in the environment and this could lead to other actions that compromise a system.

Perhaps, one of the most interesting remarks, is that looking back at the research context in Chapter 1.3 one can observe that invariants at runtime are not necessarily violated due to the interaction of programs but that there are cases such as "Re-entrance safety" where programs can be coerced into a state by the legitimate actions of users. A legitimate action could constitute manipulating a program by accessing that program using an interface and when "Signals and Events" are involved (Chapter 4) just triggering a handler at an inappropriate time may be sufficient to violate some invariant and thereby compromise a software package.

For that reason, it is necessary to be able to break down a program into its smallest components and to be able to study code by segments rather than as a whole program. As a practical comparison, one could break down a program at the level of assembler instructions, or machine language, down to the lowest level and study the program at the level of instructions. Instead of doing that, a more elegant way to study attack patterns that rely on invariant violations would be to make use of some logic reasoning that would be able to break down chunks of code into their lowest components.

This is the case of traces [23], originally used for Petri nets, that are able to illustrate a snapshot of a program. In this thesis, traces are applied to security and are used to reason about interactions on the lowest levels - whether these interactions occur between concurrent programs, or whether the interaction takes place between the user and programs.

Traces are defined formally as a sequence of state changes:

$$t = (\sigma_1, \sigma_1')(\sigma_2, \sigma_2') \dots (\sigma_n, \sigma_n')$$

where $\sigma_1, \sigma_2, \dots, \sigma_n$ represent starting states and $\sigma_1', \sigma_2', \dots, \sigma_n'$ represent ending states so that every end state follows a starting state ($\sigma_i' = \sigma_{i+1}$). The length of a trace is variable, so that any number of state pairs may be contained within a trace. In practice, this allows us to study a vulnerability in a program independent of the rest of the code.

Traces can be composed sequentially, in order to describe a larger code segment. For example suppose that we have a trace:

$$t_1 = (\sigma_1, \sigma_1')(\sigma_2, \sigma_2')$$

and a trace:

$$t_2 = (\sigma_3, \sigma_3')(\sigma_4, \sigma_4')$$

We can compose the trace $t_1$ and $t_2$ and name that resulting trace $t'$ such that $t' = t_1 \cdot t_2$ iff $\sigma_2' = \sigma_3$ and additionally the trace $t_1$ is a terminating trace. Although, the concept of "terminating trace" is not specifically used for the taxonomy, a terminating trace denotes a trace that has successfully completed execution. This notion is helpful in order to build proofs.

We use two binary predicates, the rely $R$ which represents a set of conditions that a state-change made by a trace $t$ depends on and the guarantee $G$ which represents the set of conditions that the state-change will guarantee to the environment. For example, a process might require that a shared resource does not change during an operation. $R$ and $G$ represent informally a mutual contract between the state of a process and the environment.

For example, let there be a trace:

$$t = (\sigma_1, \sigma_1')(\sigma_2, \sigma_2') \ldots (\sigma_n, \sigma_n')$$

We write $t$ **rely** $R$ iff

$$(\sigma_i, \sigma_i') \models R \text{ for } 0 \leq i \leq n$$

and $t$ **guar** $G$ iff

$$(\sigma_i, \sigma_i') \models G \text{ for } 1 \leq i \leq n.$$

We say that, if a trace's rely $R$ is respected by the environment and that trace offers some guarantee $G$ to the environment, then that trace is a well-behaving trace. In other words, if a traces's rely $R$ is respected by the environment and furthermore that trace offers the guarantee $G$, then that trace complies with the specification of the program.

By sequentially composing traces, we can specify that a rely or guarantee has to be maintained for the duration of the composition of traces. Suppose that we have two traces $t_1$ and $t_2$ and that we compose them together such that $t' = t_1 \cdot t_2$, then we say that $t'$ **rely** $R$ iff:

$$t_1 \textbf{ rely } R \text{ and } t_2 \textbf{ rely } R$$

Similarly, for the guarantee G, we say that $t'$ **guar** $G$ iff:

$$t_1 \textbf{ guar } G \text{ and } t_2 \textbf{ guar } G$$

Intuitively, this means that if we have a trace $t'$ that describes a sequential composition of multiple traces $t_1$ and $t_2$, then the trace $t'$ relies on a given rely R, respectively offers a given guarantee G to the environment if and only if all the traces in the composition rely on the same rely R and offer the same minimal set of guarantees G to the environment.

Each trace may have a pre- and postcondition [43] which are similar to assertions. In terms of predicates, if a precondition $P_1$ holds before a trace $t$, and if $P_2$ is the postcondition for the trace $t$, then $P_2$ will hold after $t$ takes place.

Another concept we use in our taxonomy is called "stability". Stability allows us to define a set of requirements imposed on a trace. We use the pre-post operator $\triangleright$ that takes states satisfying a precondition $P_1$ ($s \models P_1$), to some states satisfying a postcondition $P_2$ ($s' \models P_2$). If that transition occurs under certain conditions, represented by the rely $R$, we write $P_1 \triangleright P_2 \subseteq R$ which represents the stability condition that a transition relies on. $P_1$ and $P_2$ are unary predicates while the rely $R$ is a binary predicate representing a set of pairs.

For example, suppose we have a trace:

$$t = (\sigma_1, \sigma_1')(\sigma_2, \sigma_2')$$

If the precondition for the trace $t$ is $P_1$ and thus $\sigma_1 \models P_1$, and if we write $P_1 \triangleright P_2 \subseteq R$ then it must be that $\sigma'_2 \models P_2$ or else the rely R for the trace has been violated.

We name this "stability" because it allows us to specify some invariant that can be maintained for the duration of a trace. This is precisely the logical framework that is needed to reason about what we have mentioned in the previous chapters about smaller invariants within a program. Since many commands are seldom atomic, it is possible for an attacker to inject its own trace. For example, suppose that for the duration of one or more commands some shared variable must remain unchanged. In this case, for those commands, the shared variable becomes part of the rely and the stability condition would specify that the variable must not be changed for the duration of that trace.

To understand stability in terms of programming, consider the following program illustrated in Figure 5.2 that is meant to compute $2^{10}$ and print out the result.

```
double result = 1;
extern fpow(int b, int e) {
        do {
                if(e & 1) result *= (double) b;
                e = e >>1;
                b *= b;
        } while(e);
}
void sighandler(int signal) {
        result = 0;
}
int main(void) {
        signal(SIGHUP, sighandler);
        fpow(2, 10);
        printf("%.0f\n", result);
}
```

**Figure 5.2:** This program binds via signal identifier `SIGHUP` to the signal handler `sighandler`, then starts to compute the $2^{10}$ using the `fpow` function. It does that by storing the intermediary result in the global variable named `result`. After the function `fpow` terminates, the result is printed out.

In the case illustrated in Figure 5.2, we can say that if the precondition for the program is that `result = 1`, then the program relies that a signal will not be sent and if that rely is satisfied then the postcondition of the program will be that `result = 1024`.

In terms of predicates, suppose that the precondition of the program is $P_1 = \{\text{result} = 1\}$ and that the intended postcondition of the program is $P_2 = \{\text{result} = 1024\}$. The stability condition for the program is that $P_1 \triangleright P_2$ which is included in the rely $P_1 \triangleright P_2 \subseteq R$. The specification we give is that the rely R has to be maintained for the duration of the program. If the rely R is not maintained for the duration of the program, then $P_2 = \{\text{result} = 1024\}$ may not hold after the program terminates.

In terms of traces, the program can be described as a sequential composition of several traces:

$$t' = t_1 \cdot t_2 \cdot \ldots \cdot t_n$$

The stability condition $P_1 \triangleright P_2 \subseteq R$ means that at the beginning of the program, the first state of the trace $t'$, and thus the first state of trace $t_1$ satisfies $P_1$ then the last state of the trace $t'$, and thus the last state of trace $t_n$ **must** satisfy $P_2$ in order for the rely R to not be violated. Where, in the example illustrated in Figure 5.2, $P_1 = \{\text{result} = 1\}$ and $P_2 = \{\text{result} = 1024\}$.

This tells us that for any prefix of the trace $t'$, the rely R must be granted by the environment. In practice that means that for any intermediary step during the calculation of $2^{10}$, the value of the global variable `result` must not be changed by an external program by sending a `SIGHUP` signal.

It is also interesting to observe from Figure 5.2 that the precondition for the program is indeed $P_1 = \{\text{result} = 1\}$ and if we had a trace before $t'$, named $t''$ then we can only compose $t''$ and $t'$ as in:

$$t''' = t'' \cdot t'$$

if and only if the postcondition of $t''$ allows the variable `result` to have the value 1. For a counter-example, consider a case where a concurrent thread would have access to the variable `result`. If that thread sets `result` to anything else other than 1 before letting the program run, then the postcondition of $t'''$ would not be $P_2 = \{\text{result} = 1024\}$ even though $t'$ is the last trace and the program terminates.

We do not reason about what the outcome will be if the rely and guarantee have been violated, but in the example illustrated in Figure 5.2 one can observe that the outcome of violating the program's rely, will be that the value of `result` may be 0, which is not the intended behavior of the program.

We also cannot force a trace to be well-behaved, we can only offer a specification of a program. If the program respects that specification, then we can say that the program is safe. We do not reason about what will happen after the safety of the process has been compromised because that may lead to a wide range of consequences. Instead, we offer a minimal set of required conditions and restrictions so that the program may be well-behaved and safe.

Using this fine-grained semantics, we are able to pin-point the exact reason for a misbehaving program. If a trace consists of a number of state changes, then all prefixes of that trace rely on $R$ and must also guarantee $G$ to the environment. In the event that certain traces have to be re-wired or even eliminated, the program may not offer the same set of guarantees to the environment and might violate the program specification.

The process of studying a certain vulnerability of a software package cannot be automated just by using traces. Traces provide a high-level overview of a code segment or a program but they require the rely and guarantee sets to be clearly defined. This cannot be automated unless one already has a thorough analysis of what the code expects and offers to the environment at each step. Although that may not be feasible in practice, given such a specification of a program, it would be possible to automatically check whether that specification is respected and that invariants are maintained throughout the program.

### 5.2.1 Traces and Swimlanes

We represent traces using swimlanes such that each lane, depicted by a vertical grey bar, represents a process lane. Each trace is also annotated with a precondition and a postcondition. Control flow may be observed by following the connective arrows from top to bottom that represent the concurrent interleaving between traces. Whenever some interleaving occurs, the connective arrows cross over the red lines which represent the "trust boundary". The trust boundaries are a reference to the confused deputy problem [42] where some process influences another indirectly by causing some control flow branching.

Based on the trace semantics introduced in the previous chapter, a swimlane representation of the program is able to show only an isolated snapshot of an attack. It does not represent an image of the whole program. That is, in a swimlane diagram, the top and bottom do not represent the full length of the program but rather a slice of the code that is affected by the vulnerability to be studied.

For example, suppose we have a trace $t$ and a trace decomposition $t = t_1 \cdot t_2$. Trace $t_1$ has precondition $P$ and postcondition $Q$ and belongs to a process, while the second trace $t_2$ has the precondition $Q$ and the postcondition $S$ and belongs to a different process. We can represent the composition of the two trace $t_1 \cdot t_2$ by using two swimlanes, each one on different swimlanes belonging to one process as can be seen in Figure 5.3.

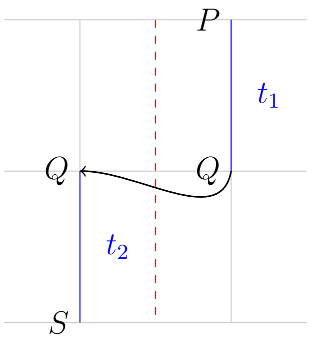


**Figure 5.3:** Example of a two-process interleaving of two traces in sequence. The postcondition of the trace $t_1$ becomes the precondition of the trace $t_2$. Code-wise, this is an example of sequential execution where the result of a former action may be observed in the subsequent actions.

Similarly, we can also represent two or more parallel traces running concurrently. Suppose we have two traces $t_1$ and $t_2$. Their parallel composition $t_1 \bowtie t_2$ can be represented in the swimlane notation as two threads on two different lanes on the same level. In this case, $P$ is the precondition for trace $t_1$ and the postcondition is $Q$. Symmetrically, for trace $t_2$ we have precondition $R$ and postcondition $S$ as can be seen in Figure 5.4.

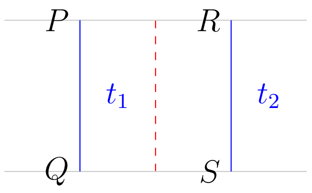


**Figure 5.4:** Example of two traces, $t_1$ and $t_2$ representing two processes or threads executing commands in parallel. In this particular case, there is no interleaving in what regards control flow and the two threads are independent.

The dashed line separating the processes in Figure 5.3 and Figure 5.4 represent trust boundaries which do not necessarily separate processes. There may be cases, for example, where two traces belong to the same process but due to the interaction the boundaries are still there and observed implicitly as a trust between code segments. Whenever an interaction takes place between processes, these boundaries are crossed either indirectly by triggering some action or directly by altering some shared resource.

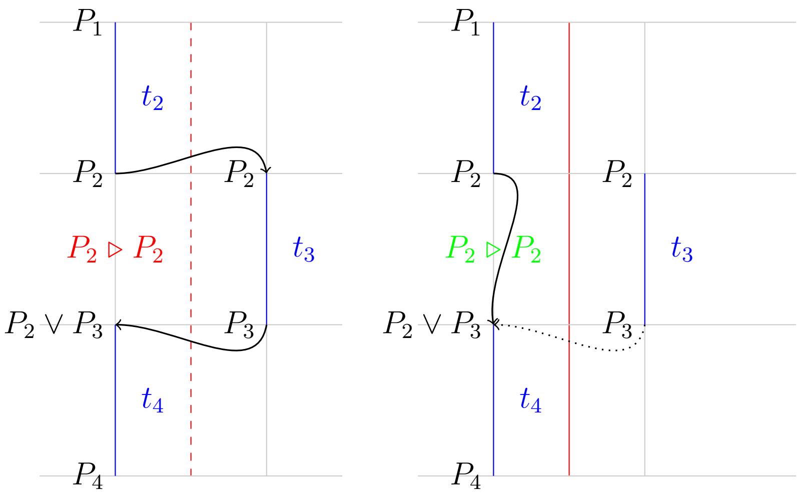


**Figure 5.5:** On the right, the stability condition $P_2 \triangleright P_2$, provided by the program's specification is respected so that under safe circumstances the postcondition of the trace $t_2$ must be $P_2$. On the left, the stability condition $P_2 \triangleright P_2$ is not respected and the attacker injects its own trace $t_3$.

Sometimes, given a program running in a concurrent context, it is necessary to emphasize that a transition from a postcondition to a precondition must conform to some set of rules. This is useful, for example, in vulnerabilities where control flow or a shared resource has been altered by an attacker. A stability condition $P_2 \triangleright P_2 \subseteq R$ may be imposed on the transition from the precondition $P_2$ to the postcondition $P_2$, or, in this case the postcondition $P_3$.

In our diagrams, a stability condition is placed between two traces on the same swimlane indicating that the transition from a postcondition of a trace to the precondition of another trace is governed by a relation $R$. In the swimlane representation (Figure 5.5), $t_2$ and $t_4$ represent two traces belonging to the process on the left swimlane and $t_3$ is a trace belonging to an interfering process on the right swimlane. The stability condition here is that $P_2 \triangleright P_2 \subseteq R$ which means that the predicate $P_2$ belonging to the process containing the traces $t_2$ and $t_3$ must be stable and included in the rely. The traces, and the predicates are presented in an abstract form; however, in our examples we instantiate the traces and the predicates individually on a case-by-case basis, given an example of a vulnerability.

We know that an attacker trace can only interfere by violating the stability conditions. If the attacker disregards the stability conditions, then we know that the safety of the process has been compromised. We do not study what will go wrong after the safety of the process has been compromised. Instead, we are able to offer the minimal required conditions so that the program may be considered safe.

The swimlane representation of traces is inspired by UML [33]. There have been similar advances in computer security, mainly by Jan Jürjens proposing an extension to the UML language, called UMLsec [49] which adds semantics for secure information flow, role-based access control, and extensions for data security. The flow of diagrams is inspired and similar to message sequence charts [4] - where processes are able to exchange messages at different times. The message passing structure is present in the swimlane diagrams implicitly due to the fact that the reasoning is applied at the level of what processes expect at any given time. In that sense, it may be the case that there are no messages exchanged between processes but due to the environment that is shared between them, some process may rely on some invariant that can be influenced by a concurrently running process. In the scope of flow-control, we do not need a rich structure to represent concurrent processes and properties because in this case, the passed data or the channels for exchanging information are irrelevant for the type of attacks that we analyze.

## 5.3 Program Refinement

Refinement [32] means to apply a series of transformations to a program so that program flow is changed into a more favorable one, either having optimizations in mind or, in the article's scope, code safety with the explicit condition that the behavior of the program has been maintained. A useful refinement in our taxonomy is a refinement that eliminates unsafe behavior.

Compared to refactoring, Roberts mentions in his thesis that "informally, a refactoring is correct if the program behaves the same after the transformation as it did before the transformation" and, further on concludes that "a refactoring is correct if a program that meets its specification continues to meet its specification after the refactoring is applied". Although Robert's thesis is not focused on concurrency, this goes well and hand-in-hand with what we we do when we rearrange traces to avoid a vulnerability, in the sense that we just refine or adapt that definition to a context which includes concurrency. Just as Roberts has a specification, we have our own specification for refactoring programs, which includes concurrency concepts such as rely and guarantee. The difficulty lies in the fact that given unknown dependencies between the environment and a program, whenever we refactor a program in a concurrency context, the resulting refactored program will never preserve the entire behavior (including, in our case, the *outbound* behavior) of the original program.

Since we cannot assume anything about what the environment depends on that should be provided by our program, we may at most say how much a refactoring has altered the original program. In doing so, we at least provide some information to the environment about what the refactoring has changed internally and the environment would have to adapt to those changes. We are changing the specification of the program implicitly when we apply a refactoring in a context where concurrency should be taken into account.

Since a program may be represented as a series of state changes, those changes may be observed at runtime. A refactoring of the code will influence that series of state changes. For example, given a trivial program that just declares a variable of an integer type and then assigns a value to it and terminates, a refined version of that program may satisfy the "overall functionality" of that program. However in doing so the refinement might alter the execution leading to the final result. For example, by first assigning 1 to the variable and then incrementing the necessary amount of times in order to reach the value in the original program. Under the assumption that compiler optimizations are turned off, the two programs may look differently on machine level with several intermediary execution steps that are different between the two programs.

In this case, the original program may have a declaration and an assignment whereas the refactored counterpart may have a series of assignments. Although a trivial example, when dealing with concurrency and shared resources, in the background Section 5.2 we have seen that state-pairs may offer a guarantee and depend on a rely. The problem is that depending on how the program is refactored, we might not be able to offer the same set of guarantees to the environment. Due to the fact that some other concurrently running program may depend on the intermediary steps which have to be factored out for the sake of security.

Summing up, when code-level changes are made by a refactoring, some states may have changed. That is what we call a *partially legal* refactoring which is what we use for the "Signals and Events" branch of our taxonomy. We call it partially legal because by Robert's definition, we are preserving the overall behavior of the program but we are removing a part of it so that the specification of one program is not identical to the refactored result.

In the solution we offer to the Zalewski signal handler exploit, the signal handler, after we refactor the program, will not be able to be called multiple times as it was the case in the original program. In that particular exploit and given how it is originally written, the signal handler will always trigger a double free corruption. However, if the situation were different, some concurrently running program may have depended on that signal handler to run multiple times. When we refactor a program we may damage the relationship between our original program and the environment. By refactoring we might solve an issue yet on a larger scale, other programs might be broken and expecting the old behavior of the refactored program.

All refactorings that change states should be considered partially legal since some part of the original program's behavior has been changed but the refined result is not identical to the original program. Thus, in a heavy concurrency context, one should be precise about how much the program has changed due to a refactoring if we the outward behavior must be preserved.

As an example of our "TOCTTOU" kingdom, the refactoring implies inserting locks to make sure that sensitive parts of code are made atomic. This particular type of refactoring would not alter the internal structure of the program at runtime too much since the refinement that needs to be applied does not eliminate already-existing states.

### 5.3.1 Program Refinement using Swimlanes

A graphical refactoring from an initial vulnerable program's diagrammatic representation to a safe program swimlane is a refactoring where the swimlanes have been changed so that the attacker's traces are prevented from interfering with the main program. On a swimlane diagram we illustrate that by fortifying the trust boundary between processes and by annotating the diagram with stability conditions. This graphical refactoring can be seen in Figure 7.8 which summarizes the transformation of vulnerability swimlane diagrams to safe swimlane diagrams.

In the example cases for "Time and State" and "Signals and Events", we have found that the following refactoring methods are the are the most relevant for a program refinement:

- Moving code around or discarding code, done by skipping commands using jumps or exceptions. This type of refinement applies to “Signals and Events”.
- Enforcing a stability condition by using locks to guarantee the atomic execution of several sequential commands that are susceptible to “TOCTTOU”.

The difference between the two types of refactoring can be observed in the swimlane representation of the original program and the refined counterpart. An entirely legal refinement will not change the structure of a program by rearranging traces (Figure 5.6) but will instead add code such that a stability condition has been enforced. For example, by encasing or wrapping vulnerable code between locks. This is the case for most TOCTTOU attacks where the internal structure of the program should not necessarily be altered in order to seal a vulnerability.

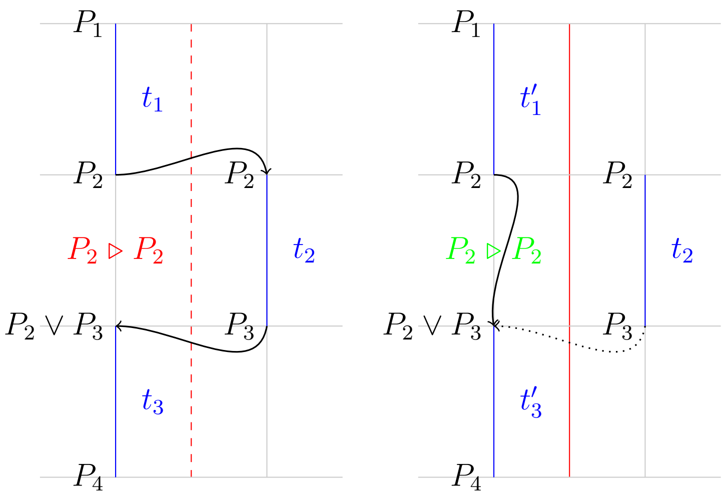


**Figure 5.6:** An entirely legal refactoring of a program is performed that alters the structure of the program on the left so that the attacker’s trace $t_2$ does not interfere with the traces of the program. The left swimlane diagram thus represents an insecure program where an attacker is able to influence a program and the right swimlane diagram represents a secure program - from the perspective of control flow.

On the other hand, we can compare the refinement needed to fix TOCTTOU problems to the refinement that is needed to fix Zalewski’s signal handler exploits. If we look at Figure 5.7 we can observe that the only change between the original program on the left and the refined result on the right is that a stability condition has been enforced to prevent an attacker to change the outcome of a transition. None of the traces that the original program had have been factored out and the original behavior has been preserved, with the exception that the TOCTTOU vulnerability has been prevented by the stability condition $P_2 \triangleright P_2$.

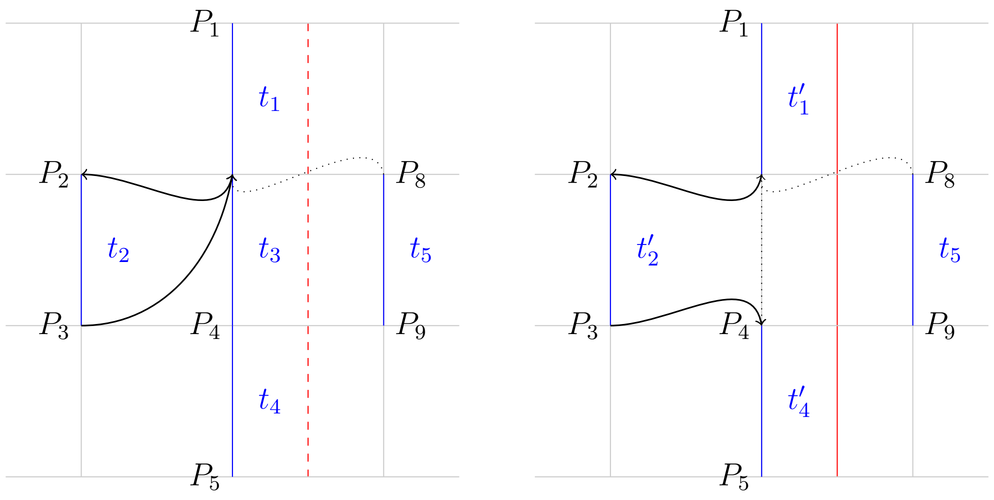


**Figure 5.7:** A partially legal refactoring of a program. The structure of the original program on the left changes in the refactored counterpart on the right so that the trace $t_3$ is factored out. By factoring out the trace $t_3$ the meaning of the program is changed which makes the refactoring a partially legal refactoring.

Compared to Figure 5.7, the partial-legal refactoring of the Zalewski signal handler vulnerability, leads to the removal of the trace $t_3$ so that the program on the left hand side is different from the refined program on the right hand side of the swimlane diagram. It is also obvious to observe in practice that by decoupling the signal handler the behavior of the program changes considerably from the perspective of control flow.

# CHAPTER 6

# TOCTTOU

A race occurs whenever two or more processes compete for a shared resource. This has the side-effect of potentially leaving one or both processes in a state which they were not designed to handle. On older hardware where there no true concurrency was available (for example, without support for threads or multicore processors), the simplest preventive measure was to check whether a resource exists before it was used. However, on modern architectures [85] operations on resources are seldom atomic and the assumptions made at the time of check may not hold at the time of use. This weakness is exploited by timed attacks, classified under "TOCTTOU" in our taxonomy.

### 6.0.2 Anatomy of a Redirection Attack

System call wrappers provide a method through which the kernel security model may be extended so that system calls may be intercepted [76]. However, in a concurrent setting, operations are seldom atomic and the operations made by system call wrappers are susceptible to timed attacks.

One example is described by Watson [83], where a shared memory segment is accessed by three different processes. A process pushes an address onto a shared memory segment, which is then read by a kernel process and validated. A third process, the attacker's process, overwrites the address after it has been checked. This is a typical instance of a TOCTTOU attack since the address that is read back by the kernel from the shared

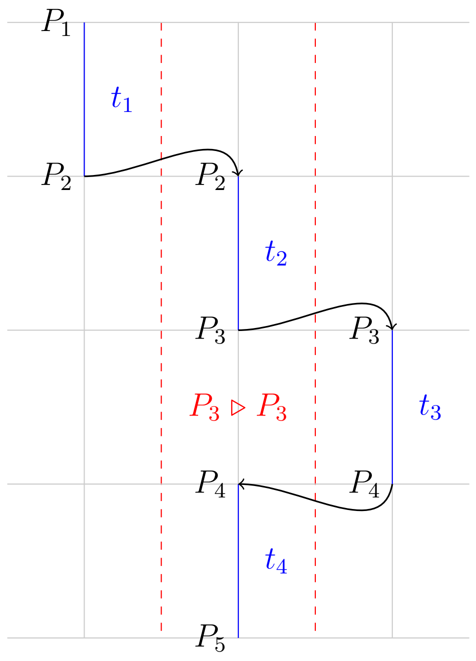


**Figure 6.1:** Three concurrent processes interleaving using swimlane representations: a description of a TOCTTOU redirection attack. The SHM segment is updated $t_1$ from userspace and passed to the kernel on the middle lane $t_2$. The kernel reads the segment ($t_2$), checks for a valid address ($P_2$) and leaves it on the segment ($P_3$). The memory segment is then read by the attacker and altered $t_3$. After the address has been abused, it is placed back on the segment ($P_4$). The stability condition $P_3 \triangleright P_3$ marked in red is not respected and the attacker is able to inject its own trace $t_4$. The equivalent code-example is given in Figure 6.3 by the concurrent interleaving of three processes.

memory segment has been altered and, most likely, made to point to a different address.

In order to solve this problem, one would specify a rely condition so that the memory segment will not be tampered with between the time of check, represented by trace $t_2$ and the time of use, represented by the trace $t_4$ as can seen in Figure 6.2.

If the attack trace $t_3$ does not alter the postcondition of the check, then the precondition for the user trace would be $P_3$. However, if the attack trace $t_3$ does interfere, the precondition for the user's trace results in the precondition $P_4$. We explicitly add a stability condition $P_3 \triangleright P_3$ which illustrates that the predicate $P_3$ should not change between the traces $t_2$ and $t_4$ thereby disallowing the interference.

On the lower layers of the taxonomy tree for this attack, the solution is to lock down a resource between the time of check and the time of use. There are several methods available, such as transactions [69] or mutex locks [12]. Another option would be to

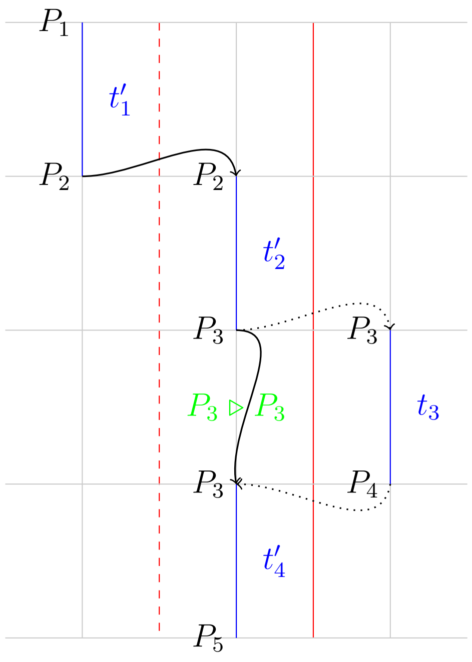


**Figure 6.2:** Compared to Figure 6.1, the stability condition $P_3 \triangleright P_3$ is enforced by locking down the shared memory segment. This does not allow an attacker trace $t_3$ to override the data placed on the segment in $t'_1$. Code-wise, in Figure 6.3 the solution would be to lock down the `add` resource.

use fractional lock permissions [8] which would still allow other concurrent processes to read from the shared memory segment. The advantage of using fractional permissions would be that, instead of locking down the resource, we would still allow other concurrent processes to read from the segment. Fractional permissions get us closer to Roberts' claims on refactoring that a program "*continues to meet its specification after the refactoring is applied*" and in this case, it is particularly important to be watchful about outbound behavior.

### Instantiating the Predicates

We take a code example based on the system call wrappers (Figure 6.3) in order to reason about the predicates. We assume that the flag `add` is a shared resource between all three concurrently running processes. The first transition is made by the user's process which sets the variable to `0`. After which, the kernel checks the variable and if it is still `0`, it sets the variable to `1`. Before the kernel uses the variable again, the attacker has already

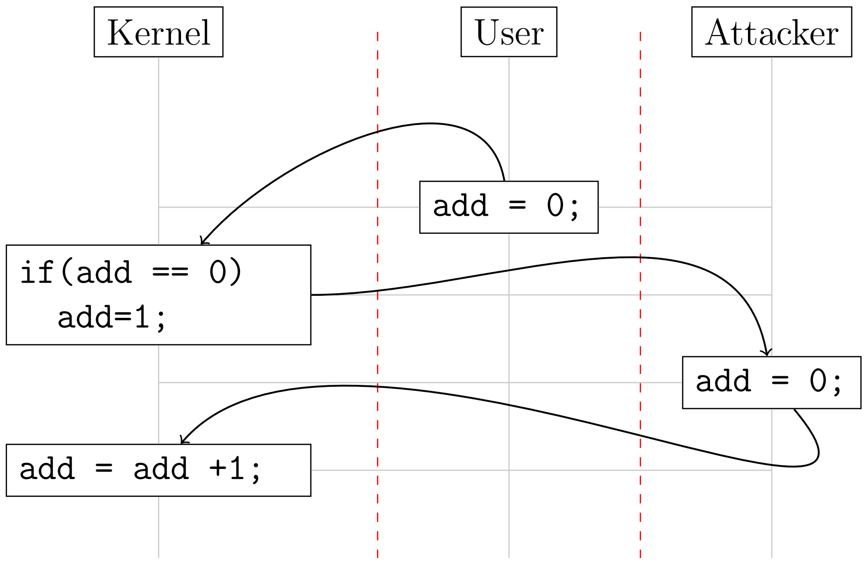


**Figure 6.3:** An interleaving of a kernel process, a user process and an attacker process showing how a variable `add` may be changed between the time of check and the time of use. The abstract swimlane diagram of this attack can be found in Figure 6.1.

set the variable back to `0`. The result is that the variable `add` now carries a value of `1` instead of `0`.

Following the pre- and postconditions described in the previous section, the user's postcondition $P_2$, becomes the precondition for the kernel's assignment and can be represented by `add == 0`, which represents a stability check by the kernel on the code-layer. After the kernel checks the variable, it sets that variable to a value of `1`, represented here by the trace $t_2$, which then becomes the precondition for the attacker's trace. We can observe that the attacker's trace $t_3$ ignores the precondition of the last trace and overwrites the variable blindly by setting its value of `0`. As a consequence, the kernel increments the value of the variable `add` during the trace $t_4$ under the assumption that the precondition is unchanged.

In this case, the attacker has no direct control over the control flow in the kernel process. However, by altering the shared resource `add`, the attacker is able to influence the outcome of the program.

### 6.0.3 Anatomy of an SQL Denial of Service Vulnerability

Concerning networking one of the challenges is to establish a protocol for allowing multiple processes to access the same resource without creating any race conditions or deadlocks [14]. Although TOCTTOU problems have commonly been noticed in UNIX software, the same concept applies to databases [58]. When inserting a row in a database one would first use a `SELECT` statement and check whether that row already exists. If that row already exists, the database returns an error. Otherwise, the database returns the number of rows modified, indicating that the insertion succeeded.

Sending `SELECT` before an `INSERT` call sounds like a feasible solution to make sure that e-mails are unique when inserting into the database. Even if the code in Figure 6.4 uses prepared statements that make command-injections less likely, in this case the same code may be called by a user that is concurrently accessing the website which could potentially lead to hijacking the first user's thread.

```
my $sql = "SELECT 1
FROM   USERS
WHERE  e-mail = $email";

my $sth = $dbh->prepare($sql);

$sth->execute();

my $val = $sth->fetchrow_array();

if ($val) {
        # a row was returned therefore e-mail in DB
        # output error-handling page
        }
else {
        # no row returned so do insert statement
        # and write normal page
        }
```

**Figure 6.4:** One way to make sure that entries are unique is to first check them using a `SELECT` statement and act upon the result by sending an `INSERT` statement if the row is not populated. The method was suggested by Paul on the perl SQL mailing-list [58], but is susceptible to TOCTTOU attacks given that other users may access the database at the same time.

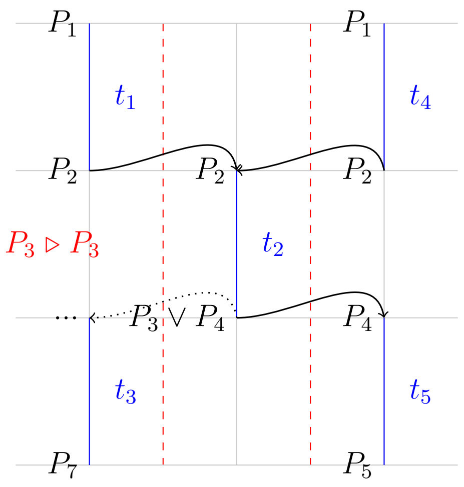


**Figure 6.5:** The interleaving of three processes: the user on the left hand side, the server executing the check in the middle and the attacker on the right. The predicates $P_3$ and $P_4$ are disjoint and represent both outcomes of the SQL statement. The trace $t_1$ belongs to a user, the trace $t_4$ to an attacker and $t_2$ belongs to the server.

The susceptible code segment in Figure 6.4 lies between the check using `if($val)`, and the `INSERT` statement that, as indicated by the comments, would be part of the `else` block. One possible solution would be to not use a `SELECT` statement at all, but instead to prefer using `INSERT IGNORE INTO` that will not cause an abort. The error-handling part, could be implemented by using the `Try::Tiny` module and displaying an error page when the entry already exists.

Assuming that an attacker is able to observe the transaction, for example as an online form submission, the attacker is able to send an `INSERT` request concurrently with the `INSERT` of a legitimate user. This may cause a denial of service for the legitimate user since the attacker's request will be handled first.

A better solution would be to use `LOCK TABLES` to ensure that the table a process has to write to is locked down [50]. In the worst case, the result would be an error sent by the database engine saying that the requested `INSERT` cannot be performed.

In Figure 6.5 we can see that the user traces (left hand side) and the attacker's traces (right hand side) run concurrently. Both the user and the attacker run exactly the same statements in $t_1$, respectively $t_4$ feeding data to the engine, represented here by the trace $t_2$. However, since the attacker is able to influence the scheduling process on the server,

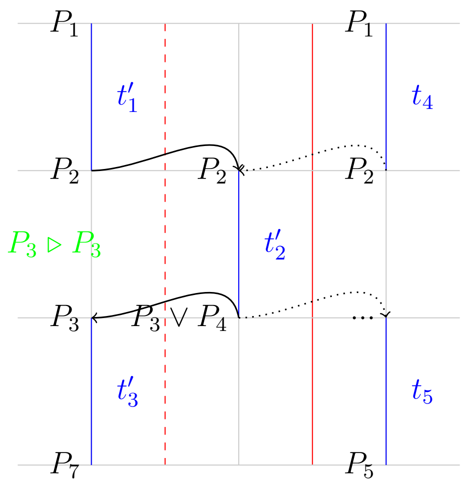


**Figure 6.6:** In contrast with Figure 6.5, when using table locking is used, we are able ensure that one single session gets handled per SQL statement. This prevents an attacker from injecting its own SQL statement before the legitimate user gets a reply from the database.

the request is handled in favor of the attacker who receives a message indicating that the request was successful $t_5$. This also has consequences for the user, because at this point the server considers that the legitimate user has already been registered and that is why TOCTTOU attacks may also be seen as a subtle form of DoS.

### Instantiating Predicates

We can instantiate $P_2$ to the check done by the SQL server that tests to see whether no other row exists for a particular primary key. However, since the attacker is able to observe the e-mail and is also able to manipulate the scheduling decision of the server, the attacker's request might get processed before the legitimate user's request is processed. After the trace $t_2$ runs, two possible responses are sent to the attacker and the user. The precondition for the user's trace $t_3$ is $P_3 \vee P_4$ which represents a check to see whether the row has been inserted successfully leading to $P_3$ or if that row already exists leading to $P_4$. Since the attacker's request has been processed first and the row has been inserted, the precondition for the user trace $t_3$ becomes $P_4$.

The stability condition illustrated in the Figure 6.5, $P_2 \rhd P_3 \subseteq R$, specifies that the user process relies on a positive result from the SQL server. In this particular case, the user's process is resilient to DoS: although the stability condition has been violated, the user's process does not lock up, waiting for a successful outcome. Instead, it receives a result from the SQL server, indicating that the row already exists and is robust enough to handle the error by taking action in $t_3$ based on the unsuccessful result $P_4$. This is also illustrated by the disjunction $P_3 \vee P_4$ which indicates that the process is able to handle both outcomes. Depending on how the client is programmed, that is not always the case and a process might deadlock and keep waiting for a positive outcome.

In practice, comparing the initial code at Figure 6.4 with the Figure 6.7 we can observe that the elegance of following the trace reasoning and locking down the tables allows us to still perform `SELECT` statements while the tables are locked down. This ensures that for the duration of the code illustrated in Figure 6.4 we are still able to perform `SELECT` statements and still avoid the TOCTTOU vulnerability.

It is the case that databases make table-locking atomic, which guarantees that there is no possible interleaving of traces between locking and unlocking the tables. An interesting consequence is that instead of refactoring the code to check for errors as the mailing-list suggests is that inserting locks would not require a major rewrite. It is difficult, if not impossible to reason about TOCTTOU statically because most concurrent interleaving takes place at runtime. However, automation in such cases may be possible if one is able to determine which traces could be manipulated by an attacker.

One may classify the vulnerability under TOCTTOU attacks and use table locking thereby making the rely of the client stronger. Or, one may choose to implement some form of access control. There may be other solutions, which just strengthens our case about building a taxonomy tree based on common weaknesses rather than distinctive solutions.

### 6.0.4 File Redirection Attacks

Filesystem attacks involve creating a softlink or altering the file between the time that a process validates it and the time when it is used. Many UNIX programs are affected [84],

```
my $sth = $dbh->prepare("LOCK TABLES");
$sth->execute();

my $sql = "SELECT 1
FROM   USERS
WHERE  e-mail = $email";

my $sth = $dbh->prepare($sql);

$sth->execute();

my $val = $sth->fetchrow_array();

if ($val) {
        # a row was returned therefore e-mail in DB
        # output error-handling page
        }
else {
        # no row returned so do insert statement
        # and write normal page
        }

$sth = $dbh->prepare("UNLOCK TABLES");
$sth->execute();
```

**Figure 6.7:** Following the suggested code from Figure 6.4, the locking variant allows for `SELECT` statements to still be used, without leaving an opportunity for TOCTTOU attacks open.

starting with a simple text editor like `vi` and ending with the upper VFS layer in the kernel.

Typical to this scenario, three parties are usually involved: the user requesting some file operation, the program initiating the check on the user specified file and the attacker replacing the file with a redirection to a different file. Even if the check and the file operation are bunched together, there is still some delay which would allow an attacker to replace the file or redirect via a symlink to a different inode.

The trace swimlanes for this attack are identical to those in Figure 6.1, which were discussed earlier in Chapter 6.0.2 and, not incidentally, we notice that the same solutions apply. A rely condition is placed by the process executing both the check and the use, in Figure 6.1 illustrated by the traces $t_2$, respectively $t_4$, so that the same preconditions apply at the time of check and at the time of use.

Symlink attacks also fall in this category and they are wide-spread on UNIX systems and targeting programs such as vi, joe, emacs but also administrative processes such as `checkinstall` and `installwatch` in earlier, incipient versions.

### Instantiating the Predicates

In a typical filesystem attack, given the swimlane representation in Figure 6.1, one could correlate the result of some check performed by the filesystem to the postcondition $P_3$. After than, an attacker replaces or redirects to a different file, possibly represented in the figure by the trace $t_3$. This will make the process write to the attacker's file in trace $t_4$, instead of the original file and would formally lead to $P_4$; the result of the attacker's trace $t_3$.

The usual scenario of a file-based TOCTTOU occurs with binaries that have `root` permissions and are able to read any file on the filesystem. A typical example can be seen in Figure 6.8 that uses `access` and `fopen`.

```
if(access("file.dat", R_OK) == -1) {
        fprintf(stderr, "No permission to access file.");
        return -1;
}
fp = fopen("file.dat", "r");
```

**Figure 6.8:** The call to `access` ensures that the program has the read permissions to the `file.dat` file. If the program does not have read permissions, then it prints out an error message and returns. Otherwise, the file is opened in read mode.

Both `access` and `fopen` operate on filenames rather than file-handles, which means that the file `file.dat` could be replaced between the call to `access` and the call to `fopen`. The usual attack pattern is that an external, concurrently-running program monitors the calls (for example, by reading file-access tables or through timing attempts) and then unlinks `file.dat` and creates a symlink from `file.dat` to a file that an attacker wants to read - for example, say password files. The consequence is that the `fopen` call will read the password file instead of the `file.dat` file.

Looking back at Figure 6.5 this can be seen in the swimlane diagrams as altering the post-condition for the outcome of `access` to $P_4$ instead of the expected $P_3$ because the attacker is able to influence the traces of the program. The solution is similar to SQL TOCTTOU and can be seen in Figure 6.6, if the filesystem permits unique access to `file.dat` for the duration of both `access` and `fopen` calls taken together.

Regardless whether we are dealing with shared memory segments or files, data is referred to by pointers and there is no explicit guarantee that the referenced data has not been altered in any way between two operations which is what TOCTTOU attacks rely on.

### 6.0.5 Refinement of TOCTTOU Vulnerabilities

In both cases, it would be sufficient to use some form of locking in order to fix the vulnerabilities. For example, some form of locking could be used to ensure that both the read and write operations are performed on the same data. In this situation, the two state pairs between the time of check and the time of use must satisfy $P_2 \triangleright P_3$ where $P_3$ is the postcondition of the check and $P_4$ is the precondition for the use.

The advantage of using locking or some form thereof, is that the specification of the program is minimally altered: all the original behavior has been preserved and atomicity is provided for transitions that are unsafe.

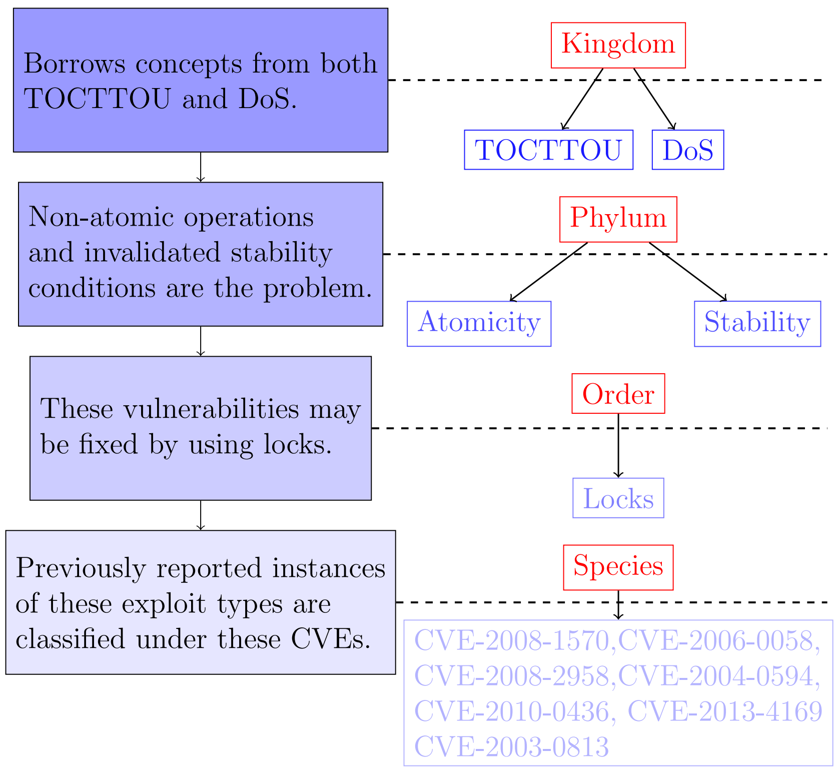
### 6.0.6 Taxonomy for TOCTTOU Vulnerabilities

TOCTTOU attacks subscribe to the same overall pattern as can be seen from Figure 6.9. Originally TOCTTOU attacks stem from the TOCTTOU kingdom but may consequentially inherit concepts from the DoS kingdom as well. In all examples of TOCTTOU that we have illustrated, the filesystem TOCTTOU attacks, Watson's syscall wrapper attack and the SQL TOCTTOU, one consequence of all attacks is that the legitimate users will not receive the expected outcome of their requests.

The impact of DoS may vary from case to case. For example, in the case of SQL TOCTTOU attacks, the DoS may take on greater proportions since the legitimate user's address will be barred from service and be considered already registered. In the case of the filesystem TOCTTOU attack, it might be sufficient in some cases, for the legitimate user requesting an operation on a file, to simply request the same operation again.

A general cure that may be applied through refactoring would be to wrap sensitive parts of code between locks. Similarly, since our taxonomy provides general solutions, the lower layers may also be treated individually. For example, in this case, the "Order" level of the taxonomy suggests "locks" as a possible remedy which may imply mutex locks, fractional lock permissions, database locks or transactions. Abstractly, no distinction is made between the different types of locks because our taxonomy attempts to use concepts to reason about vulnerabilities rather than programming language specifics which is relevant only on a case-by-case basis.

On a closer inspection, all three examples of TOCTTOU attacks resemble and inherit weaknesses from the same abstract concepts. It is only feasible to assume that the corresponding solutions are similar as well. As we can see from the taxonomy tree for this type of vulnerability, the solutions are indeed the same and differ only at code-level: filesystem, mutex or database locks.



Borrows concepts from both TOCTTOU and DoS.
Kingdom
TOCTTOU
DoS
Non-atomic operations and invalidated stability conditions are the problem.
Phylum
Atomicity
Stability
These vulnerabilities may be fixed by using locks.
Order
Locks
Previously reported instances of these exploit types are classified under these CVEs.
Species
CVE-2008-1570,CVE-2006-0058, CVE-2008-2958,CVE-2004-0594, CVE-2010-0436, CVE-2013-4169 CVE-2003-0813

**Figure 6.9:** The members of the species are the mentioned CVEs and they can be refined using locking mechanisms, the theoretical concepts being atomicity and stability, all of them showing manifestations of the concepts in the TOCTTOU or DoS Kingdom.

# CHAPTER 7

# SIGNALS AND EVENTS

The “Signals and Events” category, as a leaf of “Time and State” in the abstraction layer based taxonomy, is governed by control-flow problems with strong connections to the confused deputy problem.

Events, which are a more modern approach to signals, suffer from the same control flow weaknesses. Whenever an event is raised, a callback is installed in order to process that event. In case that callback handler modifies some shared resource then it is possible that the code within the handler may violate some stability condition on which the rest of the program relies on.

## 7.0.7 Anatomy of Signal and Events Vulnerabilities

In UNIX, signals provide a simple and efficient, if rather low-level, means of interprocess communication. Put simply, a process can cause a branch of control in another process, causing it to run a signal handler.

When control branches to signal handler, the main code of the process may see underlying state being changed by the actions of signal handlers. This situation is similar to interference by other processes. Rely-guarantee logics make this interference explicit. We adapt the idea of rely relations to signal handling.

At each atomic action of the server, traces of currently bound handlers are interleaved. This makes the handlers appear atomic from the point of view of the server, but not other processes.

We define an idealized block-structured form of signal handling in which a signal handler is installed at the beginning of the block and uninstalled at the end. It relates to `sigaction` the way exceptions related to `setjmp` and atomic `synchronized` blocks related to locking and unlocking.

In program logic, the rely-guarantee style of reasoning makes the interferences by other processes between the atomic actions of the current process explicit via binary relations.

From the perspective of rely-guarantee reasoning, there are two new features having to do with atomicity and interference: a signal handler appears atomic from the current process, but not atomic from other processes; the rely of a process changes at different points in the code, as different signal handlers may be installed and blocked.

We do not model the sending of signal explicitly. Rather, we reason about the behaviour server when receiving arbitrary signals form an arbitrary client, even a malicious one. The safety properties of the server should be maintained.

Signals resemble exceptions in that control jumps to a handler that can be installed by the program. Nonetheless, there are some radical differences. Whereas exception typically abort from the context in which they were thrown rather than returning to it, signal handlers resume control after they have run. Whereas exceptions are triggered at specific points by the code itself, signal arrive nondeterministically.

Signal handlers can have two different control flow semantics, which we call *persistent* and *one-shot*. A persistent signal can be run any number of times as long as it is installed. By contrast, a one-shot signal handler can be run at most once, as it becomes automatically uninstalled after being run the first time.

In the program logic, the difference between the two forms of signal handlers is reflected in the specifications we give for them. For a persistent signal handler, we associate an invariant $I$ to the signal that should hold before and after the signal runs. This invariant is similar to a loop invariant, where we also cannot statically determine how often the loop runs. For a one-shot handler, we associate a precondition $P$ and a postcondition $Q$ with the signal. Due to the *one-shot* semantics, we can assume that the handler encounters $P$ rather than $Q$, which would not hold if the handler could run multiple times, as it can for a persistent binding.

A signal handler is interleaved into its thread as a whole. The interference by the handler as visible from that process is given by the pre- and postcondition of the handler rather than its guarantee relation. The latter is visible as interference from the point of view of other threads, for whom the signal handler does not run atomically.

Signals are an operating system feature, which allow a program to raise a signal which will be scheduled for delivery by the kernel and, once delivered to a process, that process may use a handler to perform some operations [75]. After the signal handler runs, control flow returns to the location where the signal was initially raised. The Zalewski double free is caused by signal handler reentry by delivering two successive signals. A sample of the relevant involved code is given in Figure 7.1.

```
char *global_ptr;
void sighandler(int signal) { free(global_ptr); }
int main(void) {
        signal(SIGINT, sighandler);
        ...
}
```

**Figure 7.1:** Binding the signal identifier `SIGINT` to the signal handler `sighandler`.

Once a `SIGINT` signal is delivered, the handler `sighandler` runs and the memory referenced by pointer `global_ptr` will be freed via `free()`. Once the handler has executed, and if the platform is based on an Unix implementation, the signal is rebound to the signal handler. If a second `SIGINT` signal is delivered, the signal handler will run again and will attempt to free the pointer again which will lead to a double free memory corruption.

There is nothing preventing the signal handler to run again. Up to now, the only possible way to avoid the double free is to manually track the resources and implement checks at every step within the signal handler when a shared resource is manipulated. However, this can become quite tedious given more elaborate programs.

A typical variation of this vulnerability that works on both Windows and Unix platforms and is very common to system daemons is meant to send a message to the logs or do some form of cleanup upon termination can be seen in Figure 7.2.

```
char *global_ptr;
void sighandler(int signal) { free(global_ptr); }
int main(void) {
        signal(SIGINT, sighandler);
        signal(SIGHUP, sighandler);
        ...
}
```

**Figure 7.2:** Binding two different signal identifiers `SIGINT` and `SIGHUP` to the same signal handler `sighandler` could possibly lead to executing the same function at the same time which may further lead to data corruption if the `sighandler` function does not lock down the resources.

In this case, two different signals are bound to the same handler and although under Windows, the first delivered signal, either `SIGINT` or `SIGHUP`, is automatically decoupled, the second signal still remains bound and will still lead to a double free upon a subsequent delivery of that signal.

Similar to the the previous example, the suggested method for fixing this type of vulnerability is to simply not bind two different signals to the same signal handler. However, that does not necessarily imply that the second signal handler will not attempt to deallocate the same memory region.

In Figure 7.3 we have a swimlane diagram of the Zalewski attack which illustrates the two calls of the non re-entrant signal handler as a consequence of scheduling the delivery of the same signal twice. In order to fix the double free memory corruption, one could mark the signal handler as being non-persistent (Figure 7.4). The outcome is that once a signal handler runs, the default action `SIG_DFL` is restored for that particular signal identifier so that the signal handler will not be called a second time.

An alternative solution would be to couple exceptions [27] with signals in order to avoid using signal handlers for cleanup procedures. In effect, it would be the same as adding `exit` after `free()` in the signal handler example given by Zalewski. In that case, the rest of the operations after the memory has been freed would be discarded so that a second pass through the signal handler will not be allowed. However, that would only be

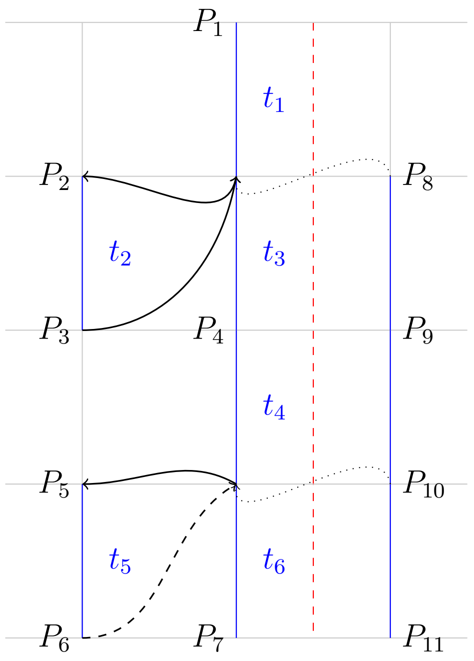


**Figure 7.3:** On the middle lane the traces $t_1$, $t_3$, $t_4$ and $t_6$ belong to a user-process. One the left lane, the traces $t_2$ and $t_5$ represent the signal handler installed by the user-process. In $t_1$ a signal handler is installed and upon the delivery of a signal, the signal handler ($t_2$) runs once and returns. The attacker's traces are illustrated on the right lane. Once the attacker delivers another signal the signal handler ($t_5$) runs again. The traces belonging to the handler are different because we reason about program state which may be different upon a second call of the signal handler.

```

if (ka->sa.sa_flags & SA_ONESHOT)
  ka->sa.sa_handler = SIG_DFL;

```

**Figure 7.4:** The linux kernel implements a programming hook which allows a signal handler to be marked as non-persistent by adding the `SA_ONESHOT` bit to the signal options.

applicable for daemons that use the signal handler as a means to clean and would not be applicable to daemons such as Sendmail which dump statistics to the system log when a certain signal is processed.

In Figure 7.5 we have a swimlane diagram where certain traces can be skipped by using exceptions. Instead of a signal handler cleaning up the memory referenced by the pointer, we have a `catch` block in trace $t'_2$. The signal handler would throw an exception and, instead of returning, control flow would jump over the traces $t_3$ and $t_4$.

The first attack following this pattern is the exploit shown by Zalewski's "Sending

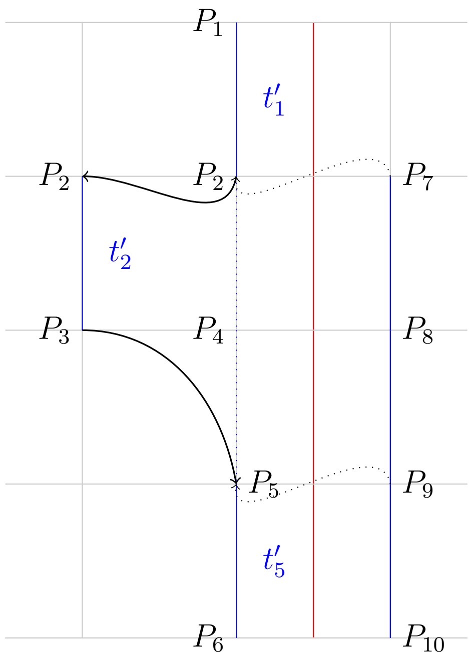


**Figure 7.5:** Contrasted with Figure 7.3, exceptions are used to discard the traces $t_3$ and $t_4$ after they have been used. This refinement allows us to bypass the undesired memory deallocation in Zalewski's signal attack by discarding the signal handler entirely using exceptions.

Signals for Fun and Profit" [89] which focuses on a vulnerability inside the Sendmail [36] as well as the WU-FTPd [40] daemon. The second is mentioned by shaun2k2's "Injecting Signals for Fun and Profit" [71]. There are other signal handler exploits which are derived from these two papers (for example, an exploit focusing on using `longjmp()` in Sendmail). Similarly, `vuftpd` versions prior to 1.2.2 contain a signal handler that uses `malloc()` and `free()` which makes it prone to an attack.

### 7.0.8 Instantiating the Predicates

We study the Sendmail (8.13.6) exploit on a lower scale by using a code-example based on Zalewski's paper (Figure 7.6). The interleaving occurs on three lanes between a handler, a process and an attacker. First, some part of the heap is allocated and referenced by the pointer `ptr` using `malloc()`. Secondly, the process performs some operations (in this case, the process just sleeps). During this period a signal is delivered by the attacker using `kill()` which schedules the execution of the processes' handler. After the first `sleep`, the handler frees up the memory referenced by `ptr` using `free()` and then sleeps again for another round. During the second `sleep`, the attacker delivers another signal which schedules the same handler again for execution. This leads to a double free so that the `syslog()` line is never reached.

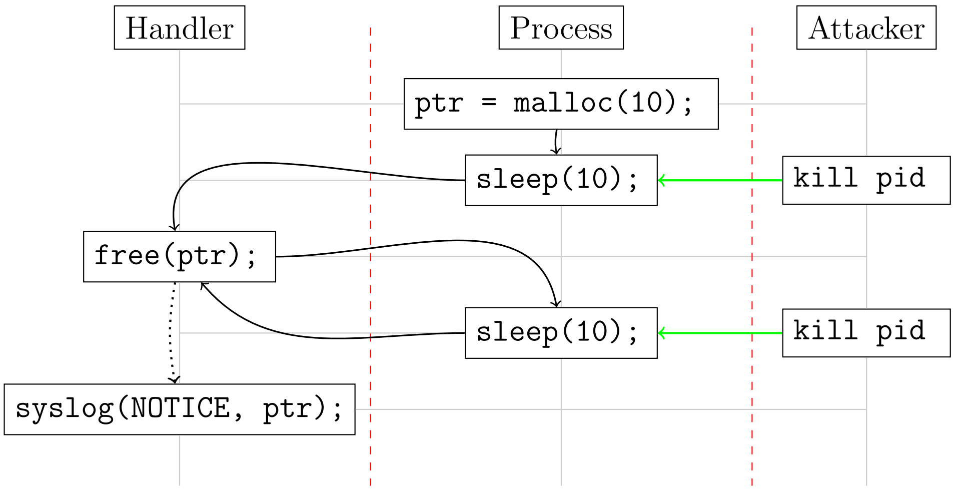


**Figure 7.6:** Signals and event exploits commonly subscribe to a typical attack pattern. The attacker induces some control flow decision in the target process by crossing the trust boundary. In that sense, the process acts as a confused deputy, being manipulated by the attacker and thus interpreting the signal delivery as a legitimate event.

We can observe that although control flow is involved, the difference between this attack and a man-in-the-middle attack is quite subtle: the attacker does not relay any messages between the process and the handler. Instead, it influences the process directly by delivering a signal which has the consequence of making the process run its own handler. In that sense, the exploit is closer to a return-to-libc attack (which is also referred to by Zalewski's paper) than a man-in-the-middle attack. The attack also differs from command injection because the attacker does not feed any particular commands to the process. Instead the process is coerced into running its own handler, choking on its own code. Nevertheless, once the double free corruption occurs, the program is left wide open and allowing the usual shellcode to be injected.

### 7.0.9 Signal Handlers and `longjmp()`

Another bug in a previous Sendmail version (8.13.6) is based on signals and uses `longjmp` inside a signal handler. The problem is described in CERT (SIG32-C) [13] and claims that `longjmp` should not be used from inside a signal handler which may lead to undefined behavior which is an explanation most likely based on Dijkstra's [26] famous article on non-local transfers.

However, the vulnerability concerning the Sendmail daemon is that a jump out of the main event loop is made at an inappropriate time. More precisely, a buffer gets allocated and deallocated in the main event loop. Between these two events a signal is delivered which will diverge control flow out of the main event loop leaving the initial buffer allocated.

Although CERT states that the problem is that by using `longjmp` from within a signal handler, the GNU C library documentation [65], documents this jump out of the main loop and is explained thoroughly in the Chapter 24.4.3 "Nonlocal Control Transfer in Handlers". Using `longjmp` from within a signal handler is not a problem in itself, however it may be that under certain circumstances such as in the Sendmail 8.13.6 daemon, a jump may be executed at an inappropriate time.

The solution to this problem is shown in Figure 7.5. The traces $t_3$ and $t_4$ from Figure 7.3 represent steps that the program skips over once the signal handler, represented by the trace $t_2$ is executed. The transitions made by $t_3$ and $t_4$ could be some deallocation routine and the precondition for the trace $t_5$, represented by $P_5$ would expect that the memory referenced by the pointer is deallocated.

The GNU C library documentation suggests that signals could be blocked by the main event loop until it is safe to run the signal handler and that case applies here very well since the jump should only be allowed to take place after the referenced memory has been deallocated. Another solution would be to run the cleanup routine, within the signal handler itself, in $t_2$ and then execute a jump instead of leaving it up to the main event loop to clean up in $t_3$ and $t_4$. Either way, the precondition $P_5$ expects that the referenced memory is deallocated before $t_5$ takes place. Alternatively, $t_5$ could perform that cleanup itself rather than leaving it up to the main event loop.

One particularly interesting thing to notice is that Zalewski's paper does not mention delivering a signal before the memory is even allocated. We can observe that the signal is delivered (Figure 7.6) after the memory has been allocated. However, the signal may very well be delivered before the pointer `ptr` is allocated using `malloc` which make the handler attempt to de-allocate a pointer with no referenced memory. One could thus time an attack to deliver a signal after a signal handler has been installed and before the global pointers are allocated.

### 7.0.10 Refinement of Interrupt-Based Vulnerabilities

When dealing with "Signals and Events", a refinement is difficult because by changing the code one may change too much of the original behavior. For example, looking at the Zalewski attack, we observe that if we were to disallow a second pass through the signal handler, then the overall behavior of the program will be changed. In a concurrency context that might not be feasible in case some other process placed in the same environment would depend on the signal handler in the other program to be able to run more than one time.

By refining a program we can thus eliminate some misbehavior due to the concurrent interleaving of processes, however that may come at the cost of altering too much of the original behavior such that dependent processes may break. It is difficult to distinguish between the behavior that must be preserved and whether it clashes with the misbehavior that must be eliminated in order to make a program safe. A good example is the TOCTTOU attack that we have studied previously where we have observed that locking down the shared memory segment would prevent an attacker to overwrite the shared data. Consequently, that would eliminate any possibility of interference but that would also limit other legitimate and concurrently running processes from accessing the segment.

Closely related, a refactoring has to make sure that the same behavior has been pre-

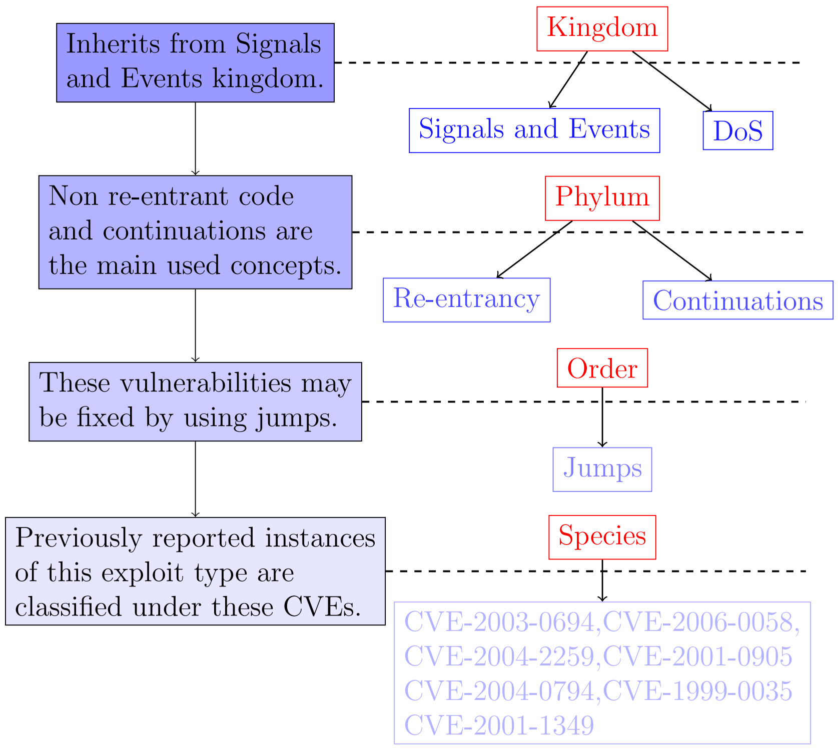

Inherits from Signals and Events kingdom.
Kingdom
Signals and Events
DoS
Non re-entrant code and continuations are the main used concepts.
Phylum
Re-entrancy
Continuations
These vulnerabilities may be fixed by using jumps.
Order
Jumps
Previously reported instances of this exploit type are classified under these CVEs.
Species
CVE-2003-0694,CVE-2006-0058, CVE-2004-2259,CVE-2001-0905 CVE-2004-0794,CVE-1999-0035 CVE-2001-1349

**Figure 7.7:** The members of the species are the mentioned CVEs and they can be refined using careful jumps, the theoretical concepts being re-entrancy and continuations, all of them showing manifestations of the concepts in the "Signals and Events" Kingdom.

served after the transformation of a program. However, when it comes to security, a refinement implicitly alters some behavior of the original program, the only condition being that the eliminated behaviors should not affect the processes that may rely on them.

Further work could formalize the process of eliminating behaviors from a program while, at the same time, preserving the necessary guarantees to the environment upon which concurrently running processes may depend on.

### 7.0.11 Taxonomy for Signal and Events Vulnerabilities

The problem with signals and events seems to be that not all handlers are not guaranteed to be reentry safe. In those cases a major refinement of the handler is required. However, if such a refinement is not feasible, one could choose to uninstall the handler after it has run. The means to do that, would be to use jumps and skip over the code that is not reentry safe [87].

| | *TOCTTOU* | Signals and Events |
|---|---|---|
| Software | sql TOCTTOU, syscall wrappers, policyd-weight, Sendmail, check-install, PHP, KDE, GDM,RPC DCOM | Sendmail, WU-FTPd, vsftpd, procmail, lukemftpd or tnftpd, ftpd |
| CVE vulnerability references | CVE-2008-1570,CVE-2006-0058, CVE-2008-2958,CVE-2004-0594, CVE-2010-0436, CVE-2013-4169 CVE-2003-0813 | CVE-2003-0694,CVE-2006-0058, CVE-2004-2259,CVE-2001-0905 CVE-2004-0794,CVE-1999-0035 CVE-2001-1349 |
| General Refinement | Enforcing atomicity using some form of locking. | Exceptions and jumps. |
| Swimlanes | Figure 6.1. | Figure 7.3. |
| Taxonomy | Figure 6.9. | Figure 7.7. |

**Figure 7.8:** We can represent the case-studies in a table so that the associated swimlane diagrams depict the same attack pattern used in all the software packages mentioned in the table.

We classify vulnerabilities (Figure 7.7) that are based on code re-entrancy issues and which may be fixed by using continuations [74] in one category, derived from the "Signal and Events" kingdom. On the lower layer, we have found several CVEs describing vulnerabilities that follow the same pattern and may be found in most common UNIX system daemons (Figure 7.8).

CHAPTER 8

# DENIAL OF SERVICE

One of the main misconceptions about denial of service attacks is that they are exclusively tied to resource exhaustion. The predominant attack pattern of denial of service attack is to disrupt a service through a battle of resources where one party overpowers the other by sending more requests than the other can handle. Good examples include fork bombs, network floods and regular expression denial of service (ReDoS) [21] which are meant to consume as many resources as possible until a server is saturated with requests and denies access to legitimate users.

On the other hand, coming back to TOCTTOU and Signals and Events attacks, a side-effect is that the service is temporarily disrupted which results in am implicit DoS for the users currently connected to the service.

The two types of attacks are different because the resource based DoS is meant to consume resources, while the control flow DoS relies on leading a program into a state that it either cannot handle or cannot escape. An useful tool to explain the difference between them is to reason using automata and contrast the two types of attacks.

A regular expression DoS (ReDoS), for example, makes the generated automaton, which is meant to process some input, increase exponentially and therefore allows an attacker exceed a reasonable quota of memory and processing power. A good example of ReDoS is `(a+)+` For a relatively small input such as `aaaa!` the number of possible paths is given by $2^2$ or a total of 16 paths. For an input of 8 characters, such as `aaaaaaaa!` the number of possible paths is given by $2^8$ which means that the automaton has 256 paths to explore. During the tests that we have performed on standard equipped machine by using the stream editor (sed), we have found that the machine can be locked-up entirely even by using very short strings. This leads to a quadratic blowup of the regular expression engine that ends up expanding with the number of input characters. In practice, modern regular expressions engines contain many additions beside the Kleene closure [57] which make the regular expression engine even more prone to be attacked.

A very similar attack pattern is XDoS [63] which relies on creating many references in XML files that makes the parsing of the file difficult due to the level of branching required. A good example of an XDoS attack would be the carefully-crafted XML file in Figure 8.1.

```
<!DOCTYPE xmlbomb [
  <!ENTITY a "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa">
]>
<xmlbomb>&a;&a;&a;&a;&a;&a;&a;&a;&a;&a;&a;&a;&a;</xmlbomb>
```

**Figure 8.1:** A XML file using an entity `a` which is referenced many times within the `xmlbomb` tags.

An attacker carefully crafts a XML file by defining the entity `a` as a long string of characters and then refers to that entity using `&a` many times within the `xmlbomb` element. The attacker then feeds this file to the parser that is forced to run through every reference and thereby consumes a lot of resources senselessly. The only viable defense is to impose limits on the depth of the paths that the parser may follow and eventually bail out if some limit is exceeded. However, this type of limitation cannot be predicted theoretically and may at best be calculated and scaled down to the resources dedicated to a process.

The same applies to fork bombs, an infamous one being the bash shell-bomb shown in Figure 8.2.

```
:(){ :|:& };:
```

**Figure 8.2:** A fork-bomb written in Bash, in compact form, with the intention of spawning child-process and thus consuming resources until no other process can be spawned. The unobfuscated version that clarifies the syntax can be seen in Figure 8.3.

When dismantled, the code amounts to a simple recursive call which forks processes and puts them in the background. An unwinded translation of the bash bomb, where we substitute the semicolon character `:` to an explicit function name `f()`, is shown in Figure 8.3.

```
f() {
  f() | f() &
}
f()
```

**Figure 8.3:** Unobfuscated version of the Bash fork bomb from Figure 8.2 showing two recursive function calls that fork child processes on every iteration.

This has a double effect because the input from recursive call to `f()` is piped into the second process using the pipe operator `|`. This is partially done in order to prevent the stub `f()` functions from returning, as well as increasing traffic over the pipe and hanging the processes. During tests we have found that even on modern Linux distributions, the bash fork-bomb manages to crash a system within seconds after it has been issued.

These types of attacks have no other purpose other than to consume resources until they exhaust the address space and thereby new processes cannot be spawned. This can be contrasted with control-flow DoS where an automaton is coerced into a state where no further transitions are possible. For both cases the *effect* is the same because they both lead to a denial of service for legitimate users, however, the *cause* for the two attacks is different. This is why DoS appears in the taxonomy as an anomaly that is self-standing by effect but can be related to other kingdoms by cause.

## 8.1 Classifying DoS using Automata

The Zalewski attack can be simplified by an automata representing transitions between states. The state $s_3$ in this case represents the state before the memory referenced by the global pointer is released. The other states $s_1$, $s_2$, $s_4$ and $s_5$ represent the states of the

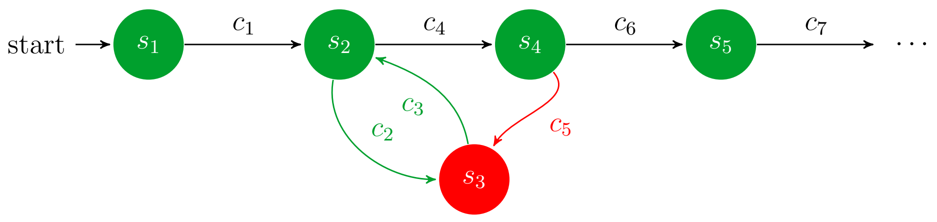


**Figure 8.4:** The Zalewski signal hander attack illustrated in terms of automata representing a DoS where a state cannot be revisited twice. This is the control-flow DoS and we can see that there is no second return from $s_3$ since the memory freed-up by the pointer has already been de-allocated. Since the program crashes if the state $s_3$ is revisited, then that state $s_3$ represents an end-state for the program.

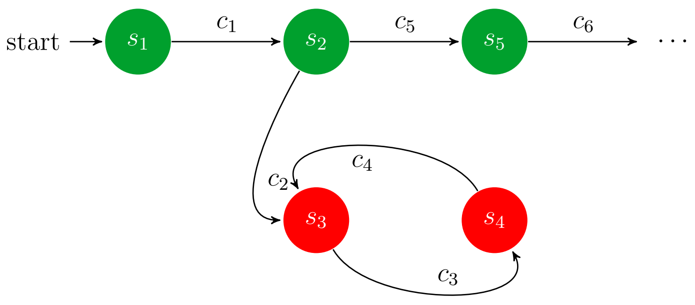


**Figure 8.5:** Another example of control-flow DoS where an automata is coerced to a branch of states from where it cannot escape. This type of DoS is similar to the Zalewski attack, with the exception that it does not have an end state. The program will spin around but it will not terminate.

main program. All transitions are made by the indexed commands $c$. Since the program can always return and execute the signal handler, the first transition from $s_2$ to the signal handler runs as expected, and the memory is freed up. However, any later return to the signal handler from $s_4$, $s_5$, etc.. Will cause the signal handler to attempt and free the same memory location (Figure 8.4).

In terms of flow control, a DoS attack that is caused by a stuck automata would be a program that deadlocks [15] waiting to acquire a shared resource. The consequence is that the deadlocked program is stuck and any legitimate requests sent to the program will be ignored. This relates to the Zalewski signal handler attack due to the fact that freeing up the memory referenced by the global pointer leads the program into a state that should not be visited more than once. It also relates to the SQL DoS due to the fact that once the attacker registers an address in the user's name, the service is lead into a state where it will refuse a new registration under the same address (Figure 8.5).

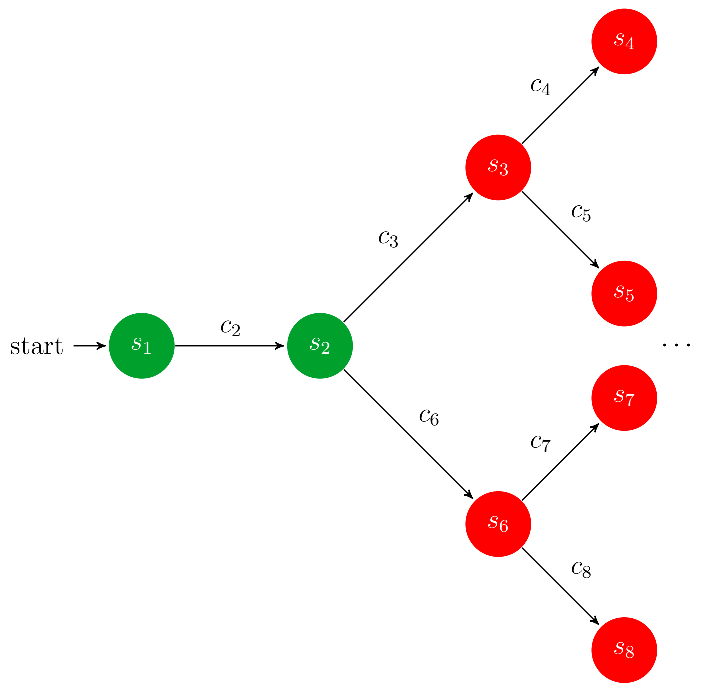


**Figure 8.6:** A branching automata that generates states, thereby consuming resources. This is a good automata equivalent of resource driven DoS attacks, such as ReDoS, XDoS, fork bombs and any type of exponential resource consumption. All of them subscribe to the same pattern where a program is made to allocate more resources in order to satisfy requests.

This is different from a resource based DoS where the program just exhausts the operating system's resources. A resource based DoS, in terms of flow control, are related to race conditions [7] where two threads spin around trying to access some resource and end up exhausting all the processing power. In terms of automata this could be seen as either increasing the number of transitions or states which consumes more computational power than the underlying system would allow. This is the case for ReDoS attacks, XDoS fork bombs and even network floods where legitimate users are attempting to access the resources that the attacker consumes (Figure 8.6).

## 8.2 Taxonomy for Denial of Service

We can build a correspondence table, showing the relationships between the various types of DoSes in order to see in which attack pattern they appear by *effect* or consequence.

The table in Figure 8.7 classifies the two types of DoS by showing the distinction

| | Resource DoS | Control Flow DoS |
|---|---|---|
| Appears by *effect* in Kingdoms | Buffer Overflows, Input Validation, Format Strings, SQL Injection, etc... | Signals and Events, TOCTTOU |
| Automata | Figure 8.6, Figure 8.5 | Figure 8.4, Figure 8.5. |
| *Cause* | Resource consumption | Unexpected control-flow branching. |
| Equivalent in terms of concurrency | Race condition. | Deadlock. |
| Prevention | Limitations imposed on resources. | Stability of predicates (disallowing transitions to states). |
| Consequence | Denial of Service. | Denial of Service and access to compromised system. |
| Instances | ReDoS, XDoS, Fork bombs, Floods. | Zalewski, Watson. |

**Figure 8.7:** We can represent the case-studies in a table so that the associated swimlane diagrams depict the same attack pattern used in various software packages mentioned in the table.

between DoSes that appear by *effect*, when the damage is consequential as well as showing the *cause* that leads to the *effect*. It is perhaps interesting to notice that resource based DoS is largely more predominant with more ties to other kingdoms than the control-flow oriented DoS. However, by influencing the branching decisions within a program, acting as a confused deputy, an attacker could achieve much more than a trivial trashing of the system.

Concerning termination, we can make the following distinction between the three automata for DoS shown in the figures:

1. The program illustrated by the automata in Figure 8.4 terminates, usually crashing the program and thereby achieving a denial-of-service attack for the users that depend on that service.

2. The program illustrated by the automata in Figure 8.5 does not terminate. It may consume resources as it spins around, it may block the system during the spin and

it may also be used to gain access. The DoS is observed implicitly as an *effect* (consequence) or it may constitute the purpose of the attack.

3. The program illustrated by the automata in Figure 8.6 does not terminate, the DoS is thereby consequential by staying alive and consuming resources.

It can observed that the second case is a mid-point case, where a denial of service may constitute the whole purpose of the attack or may be used as a staging attack for an attack meant to gain access. One extreme is given by Zalewski's signal handler attack whose sole purpose is to gain access, the DoS being a consequence (perhaps even undesirable for the attacker). The other extreme is given by the resource consumption DoS attack, with the sole purpose of making a service unavailable to others.

These distinctions grant DoS the necessary justification for being a separate kingdom. One should note that DoS may appear subclassed to many other kingdoms (Figure 8.7), as a consequence of an attack or as part of a staging attack. In case of a resource-based DoS, we also find that there are no real defenses, apart from minimizing the attack surface by imposing limits on resources. For control-flow DoS, it is frequently possible to fix a code design flaw simply by restructuring the program and eliminating the candidate point altogether.

# CHAPTER 9

# CONCLUSIONS

McGraw in his "Nineteen Sins Meet Seven Kingdoms" enumerates several broad categories mostly based on bad code practice. This type of classification is difficult to manage. For example, given McGraw's classification, we are unsure whether the signal handler exploit relying on `longjmp()` should fall into the "Time and State" category or McGraw's "Error Handling" since the signal exploit presented earlier for Sendmail is an instance of error handling. However the exploit relies on manipulating global data at an inappropriate time that would could allow us to classify the vulnerability under McGraw's "Time and State" kingdom. The DoS SQL attack we described in Chapter 6.0.3 could also fall both into "Time and State" or McGraw's "Encapsulation". Looking at the given examples, it becomes clear that an exploit can often be filed into several of McGraw's categories which makes the classification too weak. The result is that databases such as the CVE database from the MITRE corporation or the US-CERT database exhaustively enumerate vulnerabilities that are loosely classified under the most pronounced defect of a certain exploit. Finally, the "The Preliminary List of Vulnerability Examples for Researchers" (PLOVER) [18] project goes up to a list of 300 different sub-categories of attacks. Browsing such listings is only possible by using MITRE's search engine so that certain keywords such as "TOCTTOU" or "Signals" can be picked-out. There are other related databases such as CAPEC or CWE, both from MITRE corporation with a hierarchical structure that is inspired from McGraw's "Nineteen Sins Meet Seven Kingdoms" but the categories are too shallow such that the resulting taxonomy becomes too large and difficult to manage. For example, a certain vulnerability will appear as being subclassed to several defects that make the real reason difficult to comprehend. One outstanding problem is that the mentioned classifications tend to categorize the effects rather than investigating the cause for an attack pattern or, in some cases, confusing the two. As an example, one listed attack pattern is "Directory Traversal", however that frequently is an effect of improper input handling - the point being that a taxonomy built on the cause behind the attack pattern would trim down the classification considerably. An example of a shallow category is CAPEC's "Functionality Misuse" that could imply a lot of attacks. Given the vulnerabilities studied in this thesis, both the signal handler attacks and the TOCTTOU attacks could be mis-labeled under "Functionality Misuse".

Under these circumstances it is important to precisely find the common traits of various vulnerabilities in order to group them together and then possibly find a solution that may be applied to all cases which would fix most of the problems for a an entire set of vulnerabilities. If a global solution does not apply to a particular case, one could descend lower in the the taxonomy and up to the the code-level where fixes could be found on an individual basis that is particular to each software package.

One common pattern that we observe is that each studied attack inherits traits from several categories in the taxonomy. The attack that we described in Chapter 6 inherits both the fact that the attacker races the user in order to register an address, which would qualify the vulnerability as a TOCTTOU attack but it also results in a DoS because the attacker registers by using the username of a legitimate user.

We conclude that TOCTTOU attacks may be grossly solved by making use of different types of locking mechanisms. In case of filesystem redirections, one can use transactions. For the Watson system call wrappers exploit, one can use fractional permissions to allow reads while locking down the memory segment. For SQL TOCTTOU attacks, one can use table locking features available for databases.

For the inappropriate interruptions caused by `longjmp()` in Sendmail 8.13.6 one could use the signal blocking features to ensure that the referenced memory is deallocated before the rest of the program continues. One can use this technique in order to guarantee atomicity for sensitive code thereby disallowing any concurrent interleaving for certain critical segments of code. TOCTTOU attacks that depend on the execution of non-atomic operations fall into this category: asynchronous functions, non-atomic flag checks in a concurrent context and more broadly, non-atomic filesystem operations.

For signals, one may use jumps and preserve the functionality provided by the signal handlers as described in the GNU `longjmp()` manual. Thus, in Sendmail 8.13.6 one could perform the cleanup before or after the jump and still conform to the standards without losing any portability. The studied examples converge to the same issue: some situation arises during the execution of a program, when a certain precondition is invalidated by a concurrent trace. A good example is a check performed on a file pointer referencing a file. After the check, the remainder of the code works under the assumption that the file pointer has not been coerced into pointing to some other file.

Taxonomies such as "The Preliminary List of Vulnerability Examples for Researcher" [18] , "Pernicious Kingdoms" [78] and even the classification done by CERT are useful as an exhaustive listing of all vulnerabilities. It would be even more useful if the taxonomy would be built by identifying shared properties. We have summarized our conclusions for "Time and State" attacks in Figure 7.8.

One way to express tightly related vulnerabilities would be to illustrate them using in a tree structure where properties are inherited from upper layers and trickling down to the lower code level. We have shown two such trees, notably the "Signals and Events" tree (Figure 7.7) and the "TOCTTOU" (Figure 6.9) tree. The trees span from general concepts at the top and all the way down to smaller, more distinguishing traits of each studied software package on the lower leafs.

The resulting trees could also be used as a reference for software developers - perhaps comparable to a risk-based assessment that would take place during the stages of software development [59]. For example, if a developer were to know that signals will be part of the code, and that jumps are a typical source of errors in signal usage, then the developer would adapt accordingly. For the TOCTTOU tree, if a developer anticipates writing code which includes non-atomic operations, then the project could include locks preventively. For computer scientists, when dealing with an intricate attack, it becomes useful to visualize the abstract concepts behind a vulnerability rather than the small details at the code level.

The conclusion is that attack patterns filed under "DoS", "TOCTTOU" or "Signal and Events" rely on abstract concepts that are independent of language or programming API and should not be studied as being completely different on all layers of abstraction. Although the attacks are different, if we study the attacks from the "Time and State" kingdom, we can observe similarities between the attacks on layers that supersede the actual coding of the various software packages involved.

The main purpose of a signaling mechanism is to benefit from a lightweight method of notifying a process that an event has occurred. This applies to the Windows event-notification system as well, with just the slight difference that Windows events are tied to a graphical user interface rather than system daemons. However, one can observe that if an attacker is bound to manipulate flow-control, then the same reasoning can be applied to events. As we have seen, in most cases concerning system daemons, it is only necessary to indicate to a process that some state has changed so that the program can take the necessary measures and adapt.

We have examined a fair share of exploits involving signals and we believe that using traces and rely-guarantee, one could reason about the various attacks. The signals-related exploits constituting an important part of the taxonomy on abstraction layers seem to follow a common attack pattern. This allows us to group program design flaws together based on their similarities. Concerning "TOCTTOU", if we examine the attack from a control-flow perspective, we can observe that the interference between the attacker and the main program occurs at the level of shared resources which could be prevented by protecting the shared resource using locks.

For our taxonomy, on a higher level of software engineering, we find closely related research that uses the design-level of software components [3] in order to reason about the outcome scenarios that each module will have leading even to automated testing for security [2]. This is closely related to our research since composing traces together can be seen on the lower levels as a form of snapping modules together and analyzing whether certain transitions conform to a given specification. We complete that research by going one step further by using rely- and guarantee and providing a specification. That way we can pin-point exactly on the lower code-levels what transitions rely on some condition from the environment as well as to provide some assurance to the environment. The difference between reasoning and the design-level and traces is that we are able, albeit tedious, to reason about highly-concurrent processes rather than looking at the level of communication between large-scale components of software systems [60].

It is feasible to scale the taxonomy to address vulnerabilities that are not classified in the "Time and State" kingdom. The layered approach of the taxonomy offers a systematic vulnerability classifications that is able to hint to relationships between the various kingdoms and to highlight theoretical defects in software packages that other classifications do not. Additionally, compared to other taxonomies where vulnerabilities are derived from a single kingdom, the layered approach allows concepts to drift from one tree to the other. For instance, the DoS Kingdom from Chapter 8 appears in both "TOCTTOU" and "Signals and Events" taxonomy trees. One could address buffer overflows following the layered taxonomy, perhaps with "Memory violation" at the level of Phylum and "Bounds checking" at the "Order" level. The "Species" layer would thus list the numerous "Buffer Overflow" vulnerabilities gathered by other taxonomies.

Following the classification pattern for "Time and State" vulnerabilities a taxonomy can be built for "Buffer Overflows" and is illustrated in Figure 9.1. One can see that DoS is still part of the taxonomy and is still present as a consequence of the exploitation of a service. For instance, a CVE (CVE-2011-2587) listed on MITRE explains that a heap-based buffer overflow in the RealMedia demuxer in the VLC software package allows

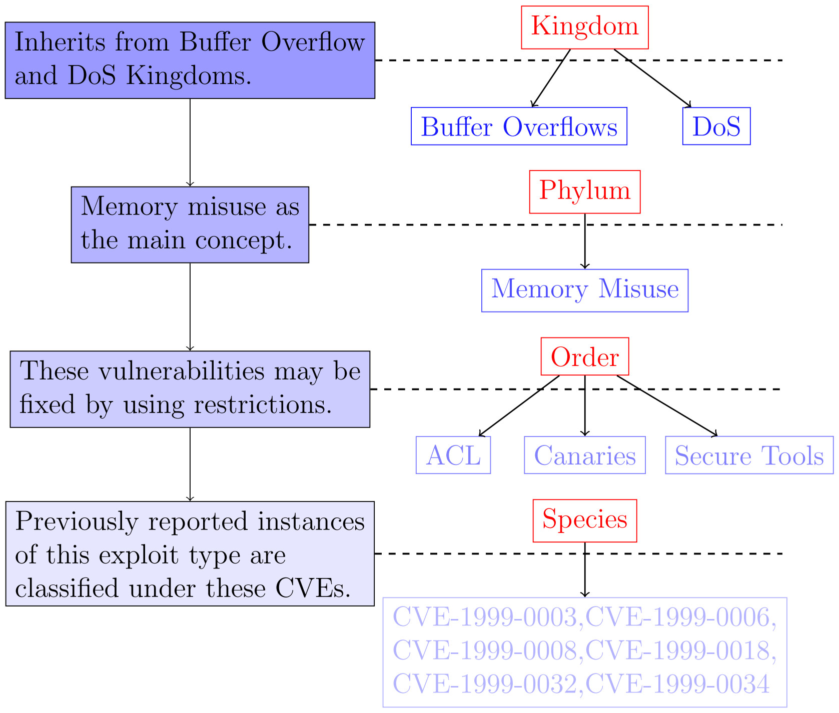

Inherits from Buffer Overflow and DoS Kingdoms.
Kingdom
Buffer Overflows
DoS
Memory misuse as the main concept.
Phylum
Memory Misuse
These vulnerabilities may be fixed by using restrictions.
Order
ACL
Canaries
Secure Tools
Previously reported instances of this exploit type are classified under these CVEs.
Species
CVE-1999-0003,CVE-1999-0006,
CVE-1999-0008,CVE-1999-0018,
CVE-1999-0032,CVE-1999-0034

**Figure 9.1:** Buffer-overflow vulnerabilities classified using the taxonomy on layers of abstraction. The DoS Kingdom is still present as a consequence. Memory misuse is the primary cause for buffer overflows and potential solutions involve minimizing the attack surface by making use of ACLs, canaries and other secure tools. The "Species" layer includes many instances of reported vulnerabilities classified as CVEs by the MITRE corporation.

a remote attacker to cause either a denial of service (application crash) or to execute arbitrary code. Usually, the common fix for buffer overflows is to minimize the attack surface by imposing restrictions by performing strict bounds checking, using canaries or making use of the tools reviewed in the "Secure Languages and Tools" Chapter 2.

Further research directions could include expanding the concept of persistent and non-persistent signal handlers. We found the one-shot binding for signals to be very useful but it would be possible to extend the concept for other attacks. A different direction could be to implement a way to throw an exception from a different program. A further research direction, could include TCP-OOB [55] signals which would make the client-server relationship of the signaling system more obvious. Traces and swimlane diagrams are also valid in a networking context. Looking at control flow, the interleaving of traces can be seen as network chatter between a server and a client.

In the signals case, a non-persistent signal handler would be a feasible fix for concurrent attacks. We can not prevent the use of asynchronous unsafe functions in a concurrent context and we can only hint that those functions are inappropriate to be used in a concurrent context. An implementation that allows the selection of the different behaviors (either persistent or non-persistent) offers a layer of reasoning that could prevent attacks. It is still up to the programmer to avoid such problems and be aware of control flow defects that may appear unexpectedly. The taxonomy on abstraction layers can only provide some insight on typical "Time and State" attacks that have previously occurred and thus allowed us to classify them.

By using traces and rely- and guarantee reasoning we are able to clearly specify conditions under which a code segment may interact with the environment. This is a fine-grain approach because it involves a well-defined specification that should be enforced upon any interleaving of the program's traces and the rest of the environment. The swimlane diagrams allow us to keep track of processes involved in the interleaving and offers an overview of states at any point during the execution of a program. Additionally by constraining the transitions with a clear specification, we can make sure that predicates and stability conditions are not violated for the duration of the program.

As a closing observation, security is a very broad domain, even when it is reduced to software security. This thesis studies attacks with a very fine-grained approach and dissects vulnerabilities into their smallest components. Since the intended audience is composed of security specialists, as well as theoreticians, it is important to highlight the fact that this level of analysis is often far superior and addresses issues on a level that may be distant to what one would encounter in a real case scenario. That means to imply that in a real case scenario, attacks take place at the level of quantity rather than at the level of quality. For example, by scanning address blocks and systematically looking for vulnerable systems. This ranges from lower-level template-based bots or crawlers that scan for opportunities, by analyzing program behavior, and by mass-probing systems. Regarding the security of servers, it may be often the case that an attack can be avoided by simply not leaking information about services. For example, it is often a system administrator's trick to change the standard access ports to services in order to minimize the attack surface. Not only does that close avenues for a potential attack, but that may also reduce the network traffic generated by attacks.

This does not advocate the notion of being obscure, in order to increase security, but it proves to be an effective measure for thwarting off arbitrary attacks. Similarly, this does not imply that elaborate attacks do not exist (in fact, they are a component of automated attacks) but that it is statistically rare that an adversary would stage an elaborate attack targeted at a particular system.

# LIST OF REFERENCES

[1] Moheeb Abu Rajab, Jay Zarfoss, Fabian Monrose, and Andreas Terzis. A multifaceted approach to understanding the botnet phenomenon. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, IMC '06, pages 41–52, New York, NY, USA, 2006. ACM.

[2] Sarah Al-Azzani and Rami Bahsoon. Semi-automated detection of architectural threats for security testing. In *Proceedings of the doctoral symposium for ESEC/FSE on Doctoral symposium*, ESEC/FSE Doctoral Symposium '09, pages 25–26, New York, NY, USA, 2009. ACM.

[3] Sarah Al-Azzani and Rami Bahsoon. Using implied scenarios in security testing. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*, SESS '10, pages 15–21, New York, NY, USA, 2010. ACM.

[4] Chien an Chen, Sara Kalvala, and Jane Sinclair. Race conditions in message sequence charts. In *In 3rd Asian Symposium On Programming Languages and Systems (APLAS05), Volume 3780 OF LNCS*. Springer, 2005.

[5] Andrew Barber. Dual Intuitionistic Linear Logic. Technical report, University of Edinburgh, 1996.

[6] J. Berdine, C. Calcagno, and P. W. OH́earn. Smallfoot: Modular automatic assertion checking with separation lgic. *FMCO*, 2006, 2006.

[7] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Comput. Syst.*, pages 131–152, 1996.

[8] Richard Bornat, Cristiano Calgano, Peter O'Hearn, and Matthew J. Parkinson. Permission accounting in separation logic. *SIGPLAN Not.*, 40:259–270, January 2005.

[9] Daniel Bovet and Marco Cesati. Signals. In Andy Oram, editor, *Understanding the Linux Kernel, Second Edition.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2 edition, 2002.

[10] Stephen Brookes. A semantics for concurrent separation logic. In *Theoretical Computer Science*, pages 16–34. Springer, 2004.

[11] BUlba and Kil3r. Bypassing stackguard and stackshield. *Phrack*, 0xa, 2000.

[12] Cristiano Calcagno, Matthew J. Parkinson, and Viktor Vafeiadis. Modular safety checking for fine-grained concurrency. *SAS*, 2007:123–134, 2007.

[13] CERT. Sig32-c. do not call longjump from inside a signal handler. `http://www.securecoding.cert.org/`, Jul 2010.

[14] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, June 1971.

[15] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, June 1971.

[16] H.V. Corcalciuc. A taxonomy of time and state attacks. In *Availability, Reliability and Security (ARES), 2012 Seventh International Conference on*, pages 564 –573, aug. 2012.

[17] H.V. Corcalciuc. A taxonomy built on layers of abstraction for time and state vulnerabilities. *International Journal of Secure Software Engineering*, 4(2):40–66, 04 2013.

[18] MITRE Corporation. The preliminary list of vulnerability examples for researchers (plover). `http://cve.mitre.org/docs/plover/`.

[19] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference & Exposition – Volume 2*, pages 119–129, Jan 2000.

[20] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks, 1998.

[21] S. Crosby. Denial of service through regular expressions. `http://www.usenix.org/events/sec03/wips.html`.

[22] Trevor Jim Dan Grossman, Michael Hicks and Greg Morrisett. Cyclone: a type-safe dialect of c. *C/C++ Users Journal*, 23(1), 2005.

[23] Willem-Paul de Roever, Frank de Boer, Ulrich Hanneman, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency verification: introduction to compositional and noncompositional methods.* Cambridge University Press, New York, NY, USA, 2001.

[24] Robert DeLine and Manuel Fahndrich. Enforcing high-level protocols in low-level software. *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 56–69, 2001.

[25] A. Diaz, I. Porpoll, and A. Crespo. On integrating posix signals into a real-time operating system. *Seventh Real-Time Linux Workshop*, 2005.

[26] Edsger W. Dijkstra. Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11(3):147–148, March 1968.

[27] Christophe Dony. Exception handling and object-oriented programming: towards a synthesis. *SIGPLAN Not.*, 25(10):322–330, September 1990.

[28] Matthew Emerson, Sandeep Neema, and Janos Sztipanovits. Safe and structured use of interrupts in real-time and embedded software. In Sang H. Son Insup Lee, Joseph Leung, editor, *Handbook of Real-Time and Embedded Systems.* CRC Press, 2006. ISBN: 1584886781.

[29] Manuel Fahndrich and Robert DeLine. Adoption and focus: practical linear types for imperative programming. *SIGPLAN Not.*, 37(5):13–24, May 2002.

[30] M. Fluet, G. Morrisett, and A. Ahmed. Linear regions are all you need. *Proceedings of the 1st ACM SIGOPS EuroSys Conference*, pages 7–21, 2006.

[31] Jeffrey Scott Foster. *Type Qualifiers: Lightweight Specifications to Improve Software Quality.* PhD thesis, University of California, Berkeley, 2002.

[32] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA, 1999.

[33] Martin Fowler and Kendall Scott. *UML distilled: applying the standard object modeling language.* Addison-Wesley Longman Ltd., Essex, UK, UK, 1997.

[34] Manuel Fhndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in singularity os. In *In EuroSys*, pages 177–190. ACM Press, 2006.

[35] Manuel Fhndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in singularity os. In *In EuroSys*. ACM Press, 2006.

[36] Jeff Gennari. Vulnerability vu nr.834865: A race condition in sendmail may allow a remote attacker to execute arbitrary code. Technical report, CERT, 2006. `http://www.kb.cert.org/vuls/id/834865`.

[37] J. Y. Girard. Linear logic: Its syntax and semantics. In J. Y. Girard, Y. Lafont, and L. Reigner, editors, *Advances in Linear Logic (Proc. of the Workshop on Linear Logic, Cornell University, June 1993*. Cambridge University Press, 1995.

[38] Sven Goldt, Sven van der Meer, Scott Burkett, and Matt Welsh. *The Linux Programmers Guide*, 03 1995.

[39] David Greenman. Serious security bug in wu-ftpd v2.4, January 1997.

[40] David Greenman. Serious security bug in wu-ftpd v2.4. `http://online.securityfocus.com/archive/1/6056/1997-01-04/1997-01-10/2`, January 1997.

[41] John V. Guttag and James J. Horning. *Larch: languages and tools for formal specification*. Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[42] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22:36–38, October 1988.

[43] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, October 1969.

[44] William F. Humphrey. Generalized callbacks: C++ and C#. *j-DDJ*, 28(3):42–43, 46–47, March 2003.

[45] G. Hunt and R. Larus. Singularity design motivation. *Microsoft Research: Technical Report*, 105, 2004.

[46] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In *In Proceedings of CRYPTO 2003*, pages 463–481. Springer-Verlag, 2003.

[47] Dipak Jha. Use reentrant functions for safer signal handling. `http://www.ibm.com/developerworks/library/l-reent/index.html`.

[48] Jan Jürjens. Umlsec: Extending uml for secure systems development. In *UML*, pages 412–425, 2002.

[49] Jan Jürjens. Sound methods and effective tools for model-based security engineering with uml. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 322–331, New York, NY, USA, 2005. ACM.

[50] Henry F. Korth. Locking primitives in a database system. *J. ACM*, 30(1):55–79, January 1983.

[51] Y. Lafont. The linear abstract machine. *Theoretical Computer Science*, 59(1-2):157–180, 1988.

[52] Craig Larmann. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process.* Prentice Hall PTR, 2002.

[53] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. *Proceedings of the 2003 Network and Distributed System Security*, pages 123–130, 2003.

[54] Patric Lincoln. Linear logic. *ACM SIGACT Notices*, 23:29–37, 1992.

[55] Robert Love. *Linux System Programming: Talking Directly to the Kernel and C Library.* O'Reilly Media, Inc., 2007.

[56] P D Magnus. *Forallx: An Introduction to Formal Logic*, volume 103. `http://www.fecundity.com/logic/`, 2008.

[57] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IEEE Transactions on Electronic Computers*, 9:39–47, 1960.

[58] Hardy Merrill. Re: Error handling. `http://www.mail-archive.com/dbi-users@perl.org/msg15388.html`, Jan 2003.

[59] Jason R. C. Nurse and Jane E. Sinclair. Supporting the comparison of business-level security requirements within cross-enterprise service development. In Witold Abramowicz, editor, *BIS*, volume 21 of *Lecture Notes in Business Information Processing*, pages 61–72. Springer, 2009.

[60] Jason R. C. Nurse and Jane E. Sinclair. Towards a model to support the reconciliation of security actions across enterprises. In *STAST*, pages 11–18, 2012.

[61] O'Hearn, P. Reynolds, and J. Yang. Local reasoning about programs that alter data structures. *Lecture Notes in Computer Science*, CSL:1–19, 2001.

[62] Hilarie Orman. The morris worm: A fifteen-year perspective. *IEEE Security and Privacy*, 1:35–43, September 2003.

[63] Srinivas Padmanabhuni, Vineet Singh, K M. Senthil Kumar, and Abhishek Chatterjee. Preventing service oriented denial of service (presodos): A proposed approach. In *Proceedings of the IEEE International Conference on Web Services*, ICWS '06, pages 577–584, Washington, DC, USA, 2006. IEEE Computer Society.

[64] PaX Project. The PaX project, Nov 2003. `http://pax.grsecurity.net/docs/pax.txt`.

[65] The GNU Project. Nonlocal control transfer in handlers. `http://www.gnu.org/s/hello/manual/libc/Longjmp-in-Handler.html`.

[66] David J. Pym. On bunched predicate logic. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, LICS '99, pages 183–, Washington, DC, USA, 1999. IEEE Computer Society.

[67] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[68] O. Ruwase and M. Lam. A practical dynamic buffer overflow detector. *Proceedings of the Network and Distributed System Security*, pages 159–169, 2004.

[69] Margo Ilene Seltzer. *File System Performance and Transaction Support.* PhD thesis, EECS Department, University of California, Berkeley, Jun 1993.

[70] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[71] shaun2k2. Injecting signals for fun and profit. `http://www.phrack.org/issues.html?issue=62&id=3`, 2000.

[72] C.J. Singer. *A history of biology: a general introduction to the study of living things.* H. Schuman, 1950.

[73] Richard W. Stevens and Stephen A. Rago. *Advanced Programming in the UNIX(R) Environment (2nd Edition).* Addison-Wesley Professional, 2005.

[74] Christopher Strachey and Christopher P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. *Higher-Order and Symbolic Computation*, 13:135–152, 2000. 10.1023/A:1010026413531.

[75] Andrew S. Tanenbaum, Gregory J. Sharp, and De Boelelaan A. *Modern Operating Systems.* Prentice-Hall Inc., 1992.

[76] Andrew S Tanenbaum and Albert S Woodhull. *Operating Systems Design and Implementation (3rd Edition).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005.

[77] Timothy K. Tsai and Navjot Singh. Libsafe: Transparent system-wide protection against buffer overflow attacks. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pages 541–, Washington, DC, USA, 2002. IEEE Computer Society.

[78] Katrina Tsipenyuk, Brian Chess, and Gary McGraw. Seven pernicious kingdoms: A taxonomy of software security errors. *IEEE Security & Privacy*, 3(6):81–84, 2005.

[79] Alan M. Turing. Intelligent machinery. *Machine Intelligence*, 5:3–23, 1969.

[80] C Viktor Vafeiadis, Viktor Vafeiadis, Viktor Vafeiadis, Matthew Parkinson, Matthew Parkinson, and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *In 18th CONCUR*, pages 256–271. Springer, 2007.

[81] J.W. Valentine. *On the origin of phyla.* American Politics and Political Economy Series. University of Chicago Press, 2004.

[82] P. Wangle, Perry, C. Cowan, and Crispin. Stackguard: Simple stack smash protection for gcc. *Proceedings of the GCC Developers Summit*, pages 243–256, 2003.

[83] Robert N. M. Watson. Exploiting concurrency vulnerabilities in system call wrappers. *Proceedings of the first USENIX workshop on Offensive Technologies*, 2007.

[84] Jinpeng Wei and Calton Pu. Tocttou vulnerabilities in unix-style file systems: an anatomical study. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, pages 12–12, Berkeley, CA, USA, 2005. USENIX Association.

[85] Jinpeng Wei and Calton Pu. Multiprocessors may reduce system dependability under file-based race condition attacks. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '07, pages 358–367, Washington, DC, USA, 2007. IEEE Computer Society.

[86] Westley Weimer. *The CCured type system and type inference.* Berkely: Computer Science Division, University of California, 2003.

[87] Jan Wloka, Manu Sridharan, and Frank Tip. Refactoring for reentrancy. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 173–182, New York, NY, USA, 2009. ACM.

[88] Chris Wright, Crispin Cowan, and James Morris. Linux security modules: General security support for the linux kernel. In *In Proceedings of the 11th USENIX Security Symposium*, pages 17–31, 2002.

[89] M. Zalewski. Delivering signals for fun and profit. `http://lcamtuf.coredump.cx/signals.txt`, May 2001.

[90] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using cqual for static analysis of authorization hook placement. *Proceedings of the 11th USENIX Security Symposium*, pages 33–48, 2002.