A Taxonomy of Time and State Attacks

Horia V. Corcalciuc
University of Birmingham, School of Computer Science
United Kingdom
h.v.corcalciuc@cs.bham.ac.uk

***Abstract*—Software classifications have been created with the purpose of keeping track of attack patterns as well as providing a history of incidents for software packages. This article focuses on one single class of such attacks, conventionally known as "Time and State" attacks. We offer a method of analyzing the anatomy of such attacks by reasoning about vulnerabilities using "swimlane" diagrams annotated with some semantics of concurrent programming, such as the notions of traces and stability. We summarize our conclusions with a taxonomy based on abstraction layers, implying thereby some form of tree hierarchy where vulnerabilities inherit properties from the upper layers and share code-level flaws on the lower layers. This approach allows us to classify attacks by what they share in common, which is different from other classification attempts.**

I. Introduction

There have been several attempts of bringing some order to security classifications, variously called "Top Ten Vulnerabilities", "Seven Deadly Sins" or "Pernicious Kingdoms" [25]. The latter classification, by McGraw and collaborators, is the most scientific one. It borrows the idea of a taxonomy from biology. McGraw predicts that more sophisticated attacks will become increasingly dangerous, such as the class of "Time and State" attacks.

This paper reasons about "Time and State" attacks and offers a method of building taxonomy trees based on layers of abstract concepts. We notice that attacks frequently exploit a theoretical concept rather than local defects in software packages. We leverage concepts from programming theory in order to make "Time and State" attacks more precise.

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 common 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 1.

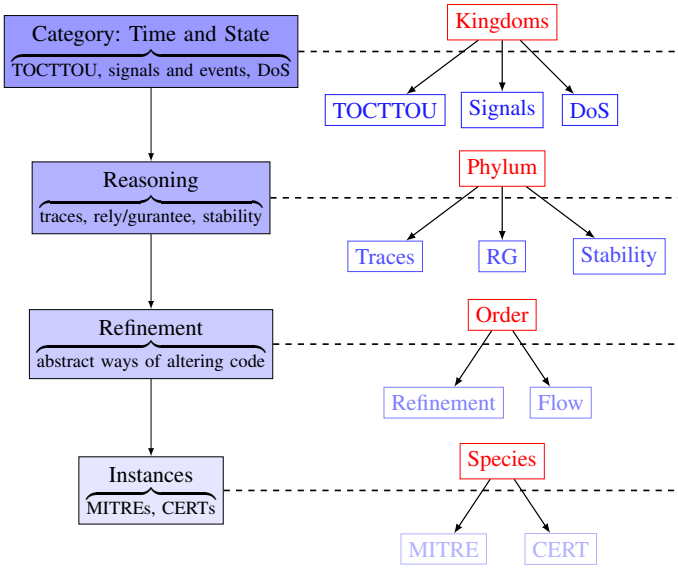

Fig. 1: Our taxonomy is annotated using McGraw's terminology. Each level describes a level of abstraction and every vulnerability can be classified by following the tree structure of a given attack. The upper layers are populated with abstract concepts such as "TOCTTOU", "Signals" and even more broadly"DoS" and reach down to lower layers where attacks distinguish themselves by local defects in a software package.

The "Kingdom" [21] rank is reserved for very high level classifications with a broad variety of descendants. The upper layers are reserved for abstract concepts which trickle down to the lower levels of the tree.

In biology, the "Phylum" rank is a grouping of organisms based on a general abstraction of structure [26]. Thus, the "Phylum" is populated by the abstraction layer holding formal concepts such as traces, states and predicates.

"Order" represents general elements of flow control and refinement [8]. In case a refinement has been performed, the code may be slightly altered but the change will be sufficient enough to change the behavior of the program.

"Species", being the last layer of the biological classification, represent individual instances of the upper

ranks. Our taxonomy uses this layer for incident reports. For example, the attacks listed in security databases such as CERT or mitre.org's CVEs.

These ranks were selectively chosen with the intent of relating to McGraw's classification and it is plausible to extend or rename them. For example, "Time and State" would be a superior layer that could be added before the rank of "Kingdoms" and named "Domain".

Denial-of-Service (DoS) is an atypical member of the "Time and State" kingdom because it is sometimes observed by effect rather than cause. Specifically, some attacks result in a DoS, although the intent of the attack was to gain access to a system. DoS shows up many times, implicitly, as a side-effect, rather than being an intended part of the attack.

The audience of this paper ranges from system administrators, programmers to security experts without going too deep into formalism.

Overview of the Paper

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 IV and signal handling attacks in Section V.

The "Background" section II explains all the formal notation that is used in the article. Although we use pictographic representations of traces, we follow-up with a section that instantiates the formalities to case-examples. 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.

The results are also summarized in the tree figures in the "TOCTTOU" section 7, "Signals and Events" section 13. We additionally offer a table in the "Taxonomy" section 12 which maps the diagrams to actual software.

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

II. Background

In order to reason about program flow, we make use of traces [6] which represent a series of state transitions allowing us to reason about control flow.

Each trace may have a pre- and postcondition [13] which are similar to assertions. In terms of predicates, if a precondition P_1 holds before a trace t_1, and if P_2 is the postcondition for the trace t_1, then P_2 will hold after t_1 takes place.

We use two additional relation predicates, the rely R which is 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.

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 the trace complies with the specification of the program.

We 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 pinpoint 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.

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.

In terms of memory, separation logic [17] also allows us to split predicates and work on disjunct parts of the heap so that we may specify that only one predicate has to be stable during a certain transition. For example, we could decompose the predicate P_1 as $P_1 \equiv P_1 * P_1'$ and the predicate P_2 as $P_2 \equiv P_1 * P_2'$. In that case, we can isolate and impose a rely condition on a slice of the heap so that $P_1 \triangleright P_1 \subseteq R$, with the predicates P_1' and P_2' being free to change.

A. 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 while connective arrows 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 [12] where some process influences another process indirectly by causing some control flow branching.

Based on trace semantics, the swimlane representation of a program shows an isolated snapshot of a candidate point in a certain vulnerable software package. Thus, the top and bottom of a swimlane diagram represents just a snapshot of some operations that are useful for illustrating a vulnerability.

Trust boundaries are used to illustrate cases where some concurrent interleaving occurs. There are cases where two traces belong to the same process. For example, signals handlers or event callbacks are not separate processes. However, if an attacker is able to influence an entity, even if that entity is part of the same program, a trust boundary can be implicitly observed.

The swimlane representation of traces is loosely derived from UML [9] from which we leverage the ability to represent control flow.

B. Usefulness of Traces and Swimlanes

In most of the cases we approach, by comparing the vulnerability swimlane diagram and the solution diagrams side-by-side, we are able to generalize and say that there exists a transformation that takes a vulnerability diagram and turns it into a solution diagram. Although we do not make the transformation explicit, we hint to the reader that the same transformation would apply to all vulnerabilities classified using the same tree-based taxonomy.

III. Refinement

Refinement [8] means to apply a series of transformations to a program so that program flow is changed into a more favourable one, either having optimisations in mind or, in the scope of the article, 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 [18] 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*. Further on, Roberts concludes that *a refactoring is correct if a program that meets its specification continues to meet its specification after the refactoring is applied*. It is difficult to be too precise about what the environment depends on; especially when multiple processes are interleaved. Thus, a refinement has to offer a minimal set of changes in order to factor out misbehaving traces without drastically changing the externally observable behavior.

IV. TOCTTOU

A race occurs whenever two or more processes compete for a shared resource. That has the side-effect of potentially leaving one or both processes in a state which they cannot 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 using it. However, on modern architectures [29] 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.

A. 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 [24]. However, in a concurrent setting, operations are seldom atomic and the operations made by system call wrappers are susceptible to timed attacks.

One such example is described in Watson's paper [27], 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 memory segment and altered, 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 we can observe in Figure 3.

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 attacker's trace t_3 does

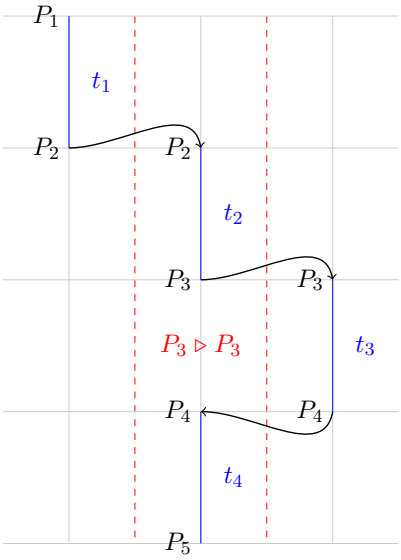

Fig. 2: Three concurrent processes interleaving using swimlane representations: a description of a TOCTTOU redirection attack. The SHM segment is updated t_1 from userspace and passed to the kernel on the middle lane t_2. The kernel reads the segment (t_2), checks for a valid address (P_2) and leaves it on the segment (P_3). The memory segment is then read by the attacker and altered t_3. After that, 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 4 by the concurrent interleaving of three processes.

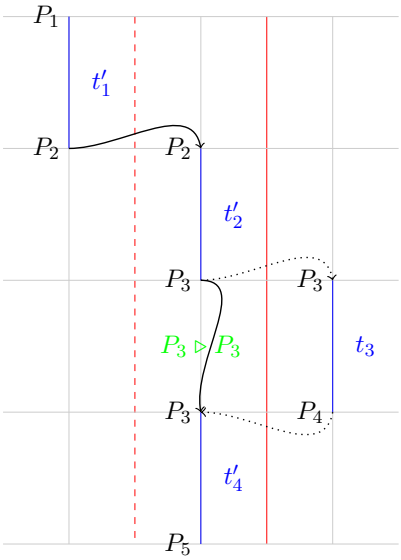

Fig. 3: Compared to Figure 2, the stability condition $P_3 \triangleright P_3$ is enforced by locking down the shared memory segment. This does not allow an attacker trace t_3 to override the data placed on the segment in t'_1. Code-wise, in Figure 4 the solution would be to lock down the `add` resource.

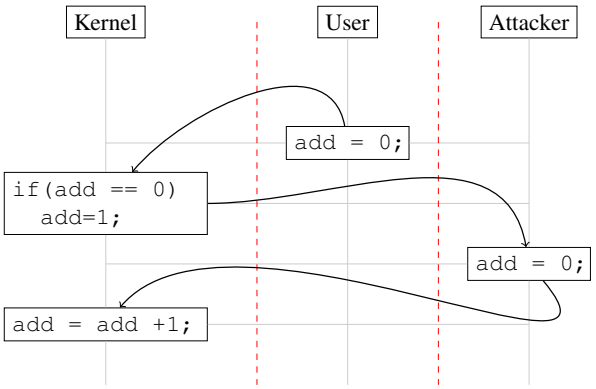

Fig. 4: An interleaving of a kernel process, a user process and an attacker process showing how a variable `add` may be changed between the time of check and the time of use. The abstract swimlane diagram of this attack can be found in Figure 4.

interfere, the precondition for the user's trace ends in 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 and thereby disallowing the interfering trace.

Based on the program's specification, we are able to prove correctness by induction which proves that, under the specification of a program, if the stability condition is not violated, then no intervention took place and the attacker has not influenced the program.

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 [19] or mutex locks [2]. Another option would be to use fractional lock permissions [1] which would still allow other concurrent processes to read from the shared memory segment. Instead of locking down the resource entirely, we would still allow other 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.*

1) Instantiating the Predicates: We take a code example based on the system call wrappers (Figure 4) 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 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.

B. Anatomy of a SQL Denial of Service Attack

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 [4]. Although TOCTTOU problems have commonly been noticed in UNIX software, the same concept applies to databases [15]. 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.

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 [14]. 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 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, 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

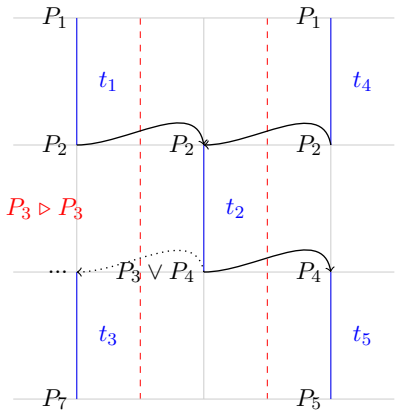

Fig. 5: The interleaving of three processes: the user on the left hand side, the server executing the check in the middle and the attacker on the right. The predicates P_3 and P_4 are disjoint and represent both outcomes of the SQL statement. The trace t_1 belongs to a user, the trace t_4 to an attacker and t_2 belongs to the server.

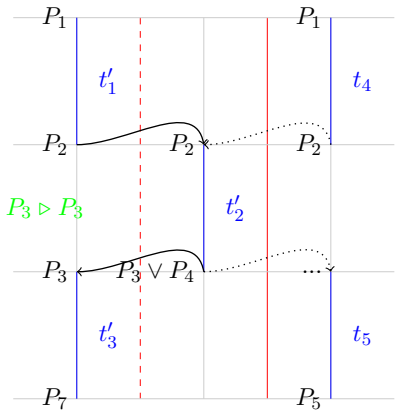

Fig. 6: In contrast with Figure 5, when using table locking is used, we are able ensure that one single session gets handled per SQL statement. This prevents an attacker from injecting its own SQL statement before the legitimate user gets a reply from the database.

that is why TOCTTOU attacks may also be seen as a subtle form of DoS.

1) Instantiating Predicates: We instantiate P_2 (on the middle lane) to the check performed by the SQL server making sure that no other row exists. After trace t_2, two possible responses are sent to the attacker and the user. The postcondition of the server's trace t_3 is $P_3 \vee P_4$ depending on 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 thereby running the statement before the legitimate user.

The stability condition illustrated in Figure 5, $P_3 \triangleright P_3 \subseteq R$, specifies that the user process relies on a

positive result from the SQL server.

We classify this vulnerability under TOCTTOU attacks and use table locking thereby making the rely of the client stronger.

C. File Redirection Attacks

Filesystem attacks involve creating a symlink or altering the file between the time that a process validates it and the time when it is used. Many UNIX programs are affected [28]. Symlink attacks also fall in this category and are widely-spread on UNIX systems targetting programs such as `vi`, `joe`, `emacs` but also administrative processes such as `checkinstall` and `installwatch`.

Three parties are involved in a file redirection attack: the user requesting some file operation, the program initiating the check on the user-specified file and the attacker redirecting to a different file. Even if the check and the file operation are bunched together, there is still some delay which allows the attacker to replace or divert to a different file.

1) Instantiating the Predicates: In a typical filesystem attack as in Figure 2, one could illustrate the result of some check performed by the filesystem by the postcondition P_3. The trace t_3 represents an attacker trace replacing or redirecting to a different file. This will make the process write to the attacker's file in trace t_4, instead of the original file and would lead to P_4 instead of the expected postcondition P_3. Data is referred to by pointers, whether we are dealing with SHM segments or files. There is no explicit guarantee that the referenced data has not been altered between operations.

D. Refinement

In both cases, it would be sufficient to use some form of locking 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 the stability condition $P_3 \triangleright P_3$.

The advantage of using locking or some form thereof, is that the specification of the program is minimally altered: most of the original behavior has been preserved.

E. Taxonomy

TOCTTOU attacks subscribe to the same overall pattern (Figure 7). Originally TOCTTOU attacks stem from the TOCTTOU kingdom but may also inherit concepts from the DoS. In all examples of TOCTTOU that we have illustrated, the filesystem TOCTTOU attacks, Watson's syscall wrapper attack and the SQL TOCTTOU,

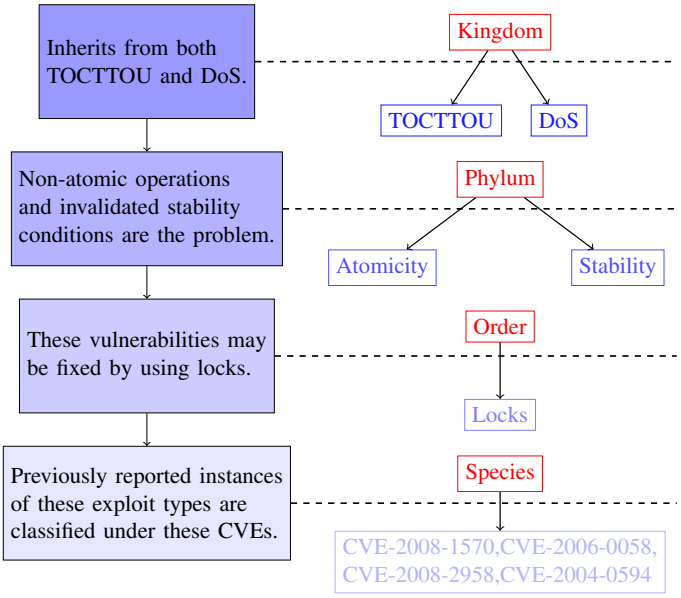

Fig. 7: The members of the species are the mentioned CVEs and they can be refined using locking mechanisms, the theoretical concepts being atomicity and stability, all of them showing manifestations of the concepts in the TOCTTOU or DoS Kingdom.

one consequence of all attacks is that the legitimate users will not receive the expected outcome of their requests. That represents a DoS for the legitimate user since the legitimate may be barred from service by the server.

The "Order" level of the taxonomy suggests "locks" as a possible remedy which may imply mutex locks, fractional lock permissions, database locks or transactions.

On a closer inspection, all three examples of TOCTTOU attacks inherit weaknesses from the same abstract concepts. Incidentally, the corresponding solutions are similar as well (Figure 7).

V. Signal and Event Handling Attacks

The "Signals and Events" category, as a leaf of "Time and State" in the abstraction layer based taxonomy, is governed by control-flow issues. Events, which are a more modern approach to signals, suffer from the same control flow weaknesses as well. When an event is raised, a callback is meant to process that event. If that callback modifies a shared resource then it is possible that the code within the callback may violate some assumption which that the rest of the code takes for granted.

A. Anatomy of Signal Attacks

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

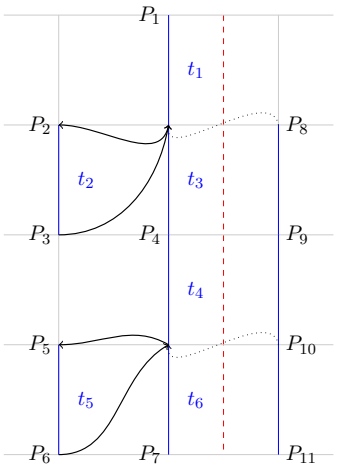

Fig. 8: On the middle lane the traces t_1, t_3, t_4 and t_6 belong to a user-process. One the left lane, the traces t_2 and t_5 represent the signal handler installed by the user-process. In t_1 a signal handler is installed and when a signal is delivered, the signal handler (t_2) runs once and then 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;
```

Fig. 9: 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.

operations [23]. After the signal handler runs, control flow returns to the location where the signal was initially raised.

The Zalewski attack relies on the fact that a non re-entrant signal handler is called twice purposefully by an attacker in order to trigger a double free memory corruption.

In Figure 8 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 9). 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 [7] 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

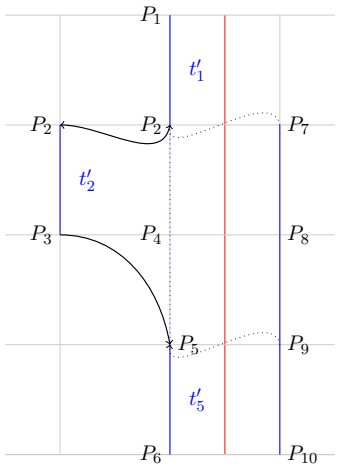

Fig. 10: Compared to Figure 8 we use exceptions and discard the traces t_3 and t_4 after they have been used. This refinement allows us to bypass the undesired memory deallocation in Zalewski's signal attack by discarding the signal handler entirely using exceptions.

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

In Figure 10 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 Signals for Fun and Profit" [31] which focuses on a vulnerability inside the Sendmail [10] as well as the WU-FTPd [11] daemon. The second is mentioned by shaun2k2's "Injecting Signals for Fun and Profit" [20]. 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 which uses `malloc()` and `free()` which makes it prone to an attack.

B. 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 11). 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.

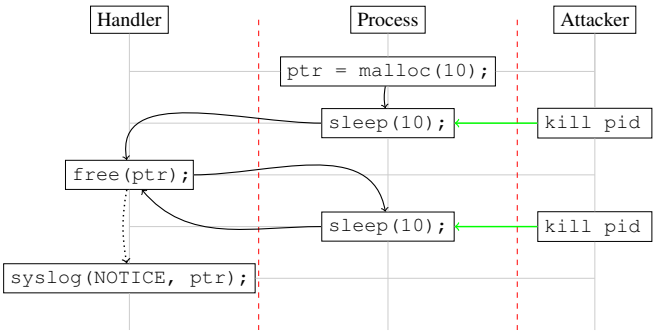

Fig. 11: Signals and event exploits commonly subscribe to a typical attack pattern. The attacker induces some control flow decision in the target process by crossing the trust boundary. In that sense, the process acts as a confused deputy, being manipulated by the attacker and thus interpreting the signal delivery as a legitimate event.

When classifying this attack, the subtle difference between a man-in-the-middle attack is subtle: the attacker does not relay any messages between the process and the handler. Instead, the attacker 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 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.

C. Signal Handlers and `longjmp()`

A different bug in Sendmail version (8.13.6) is based on signals and uses `longjmp()` inside a signal handler. The problem is described in CERT (SIG32-C) [3] and claims that `longjmp()` should not be used from within a signal handler because it may lead to undefined behavior.

The essence of the vulnerability found in the Sendmail daemon is that a jump out of the main event loop is made at an inappropriate time. Between these two events a signal is delivered which triggers a jump out of the main event loop which leaves a buffer allocated.

CERT states that the vulnerability is due to jumps from within a signal handler but the GNU C library documentation [16], documents this technique in the Chapter 24.4.3 "Nonlocal Control Transfer in Handlers". We think that using `longjmp()` from within a signal handler is not the problem but it may happen that the jump takes place at an inappropriate time which violates some assumptions that the rest of the program relies on.

The solution to this problem is shown in Figure 10. The traces t_3 and t_4 contain commands that the program skips over once the signal handler (t_2) is executed. The transitions made by t_3 and t_4 represent a cleanup routine calling `free()` to deallocate the referenced memory. A different solution would be to run the cleanup routine within the signal handler in t_2 and only then execute a jump instead of leaving it up to the main event loop to clean up in t_3 and t_4. The precondition P_5 expects that the referenced memory is released before t_5 takes place. Alternatively, t_5 can perform that cleanup itself rather than leaving it up to the main event loop.

One interesting remark 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 11) 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.

D. Refinement

When dealing with "Signals and Events", a refinement is difficult because by changing the code one may implicitly change too much of the original behavior. For example, looking at the Zalewski attack, we observe that refining the code would change the behavior of the program so that a second call to the signal handler will not be possible. In a concurrency context that might not be the best option if some other process depends on the signal handler to be called more than once.

E. Taxonomy

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

	TOCTTOU	Signals and Events
Software	sql TOCTTOU, syscall wrappers, policyd-weight, Sendmail, checkinstall, PHP	Sendmail, WU-FTPd, vsftpd, procmail, lukemftpd aka tnftpd, ftpd
Swimlanes	Figure 2.	Figure 8.
Taxonomy	Figure 7.	Figure 13.

Fig. 12: 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.

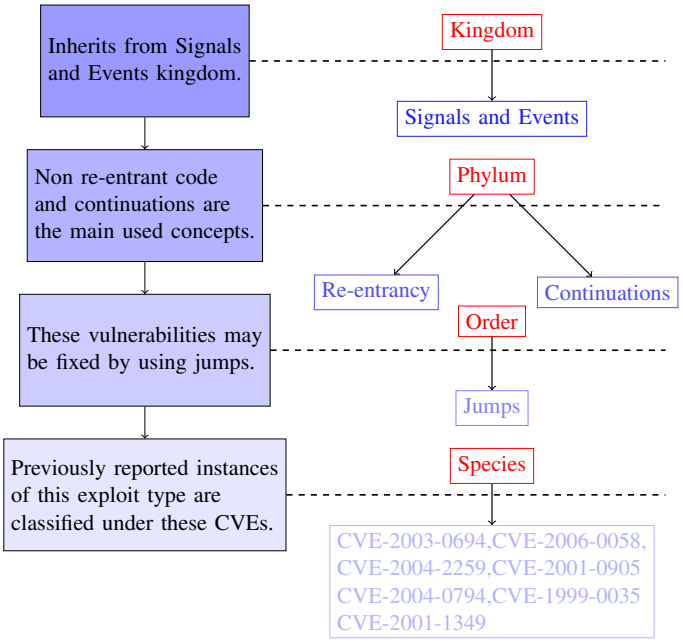

Fig. 13: The members of the species are the mentioned CVEs and they can be refined using careful jumps, the theoretical concepts being re-entrancy and continuations, all of them showing manifestations of the concepts in the "Signals and Events" Kingdom.

means to do that, would be to use jumps and skip over the code that is not reentry safe [30].

We classify exploits (Figure 13) that are based on code re-entrancy issues and which may be fixed by using continuations [22] 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 12).

VI. Other Taxonomies

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 Sendmail does just that). The DoS SQL attack we described in Chapter IV-B could also fall both into "Time and State" or McGraw's "Encapsulation". Finally, the "The Preliminary List of Vulnerability Examples for Researchers" (PLOVER) [5] project goes up to a list of 300 different sub-categories of attacks.

We believe it is important to precisely find the common traits of various vulnerabilities in order to group them together and then find a solution that may be applied to all cases which would fix most of the problems in a software package. After that, one could descend down to the lower layers, at the code-level and apply supplementary fixes on a case-by-cases basis.

The attack that we described in Chapter IV 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 themselves in the name of the legitimate user.

VII. Prevention

We draw the conclusion (Figure 12) that TOCTTOU attacks may be grossly solved by making use of locks. 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 the SQL TOCTTOU attack, one can use table locking features.

For the 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. The technique is to guarantee atomicity for sensitive code. TOCTTOU attacks which depend on the execution of non-atomic operations fall into this category: asynchronous functions, flag checks in a concurrent context and 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. All examples converge: some situation arises where a certain precondition gets invalidated. For example a check for a file pointer, and the rest of the code following that precondition works under the assumption that data has not been altered.

VIII. Conclusions

Taxonomies, such as "The Preliminary List of Vulnerability Examples for Researcher" [5] , "Pernicious Kingdoms" [25] and the classification done by CERT are very useful as an exhaustive listing of all vulnerabilities. It would be even more useful if the taxonomy would be built by identifying shared properties.

One way to do that is to represent classes of vulnerabilities as a tree structure. We have shown two such trees: the "Signals and Events" tree (Figure 13) and the "TOCTTOU" (Figure 7) tree. The tree spans from general concepts at the top and all the way down to smaller, more specific issues on the lower leafs.

The resulting trees can also be used as a reference for developers. For TOCTTOU, if a developer anticipates writing code which includes non-atomic operations, then the resulting code could include locks preventively. For computer scientists, when dealing with an intricate attack, it is very useful to easily pick out the abstract concepts governing a certain vulnerability.

Attack patterns filed under "DoS", "TOCTTOU" and "Signal and Events" represent abstract concepts which are independent of language and should not be studied exclusively on a case-by-case basis.

Acknowledgment

I wish to thank my PhD supervisor Hayo Thielecke and Mark Ryan for their guidance and support in writing this paper.

References

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

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

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

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

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

[6] 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.

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

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

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

[10] 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.

[11] 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.

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

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

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

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

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

[17] 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.

[18] Donald Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[19] Margo Ilene Seltzer. File system performance and transaction support, 1992.

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

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

[22] 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.

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

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

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

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

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

[28] 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.

[29] 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.

[30] 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.

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