SwitchBlade: Enforcing Dynamic Personalized System Call Models

Christof Fetzer
Department of Computer Science
Technische Universität Dresden
01062 Dresden
Dresden, Germany
christof.fetzer@tu-dresden.de

Martin Süßkraut
Department of Computer Science
Technische Universität Dresden
01062 Dresden
Dresden, Germany
martin.suesskraut@tu-dresden.de

ABSTRACT
System call interposition is a common approach to restrict the power of applications and to detect code injections. It enforces a model that describes what system calls and/or what sequences thereof are permitted. However, there exist various issues like concurrency vulnerabilities and incomplete models that restrict the power of system call interposition approaches. We present a new system, SwitchBlade, that uses randomized and personalized fine-grained system call models to increase the probability of detecting code injections. However, using a fine-grain system call model, we cannot exclude the possibility that the model is violated during normal program executions. To cope with false positives, SwitchBlade uses on-demand taint analysis to update a system call model during runtime.

Categories and Subject Descriptors
D.4.6 [Operating Systems]: Security and Protection

General Terms
Security

Keywords
Security, System Call Interposition, System Call Models, Taint Analysis

1. INTRODUCTION
Many desktop applications and almost all server applications maintain network connections. Network connections can make applications vulnerable to code injection attacks. For example, small coding mistakes can introduce buffer overflow vulnerabilities that might be exploitable by attackers. Dynamic taint analysis [47, 15, 34, 23, 40] detects code injection attacks by checking if the control flow has been altered maliciously (i.e., tainted) by network or other user

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

EuroSys ’08, April 1–4, 2008, Glasgow, Scotland, UK.
Copyright 2008 ACM 978-1-60558-013-5/08/04 ...$5.00.
in the original program and hence, would in general permit that these system calls are flagged as violating the normal sequence of system calls. Anomaly-based approaches use statistics on the sequencing of system calls to detect code injections. However, a smart attacker can execute additional system calls to make the sequence of executed system calls look normal [50]. In SwitchBlade, we do not use a statistical model but instead a model that describes for a given application the temporal order of system calls and the dataflow between the system calls (see Sec. 4). Currently, we only check arguments passed in registers, i.e., we are not susceptible to TOCTTOU attacks.

However, the system needs to be guarded against mimicry attacks, i.e., attacks that imitate the exact system call behavior and cross system call data flow of the program. An attacker could slightly change the arguments to system calls such that the program behavior fits the needs of the attacker without violating the system call model. We can make this attack arbitrarily difficult by using personalization (see Sec. 4.1) and randomization (see Sec. 4.2) of the system call model.

We personalize the system call model by learning a minimal system model that describes the execution of an application in a given environment. An attacker could learn the set of accepted system call sequences by analyzing the source or binary code of an application. We try to use the smallest model that describes the sequence of system calls of an application in a given environment. This is to prevent that an attacker can mimic program features that are disabled in a given environment. For example, an editor might be compiled without the feature to spawn shell scripts. Personalization will remove the system call subsequences associated with such features that are statically or dynamically (via a configuration file) disabled.

We randomize the model by inserting system calls in the application (and hence, in the model) that are not in the original program. We can increase the difficulty of mounting a successful attack by inserting dummy system calls using, for example, library wrappers or source code transformation tools. An attacker would need to be very careful to call (library) functions in the right sequence, with the right inter-system-call dataflow and with the correct backtrack. Moreover, an attacker needs to write the right return addresses in the backtrack, which are already randomized by virtual memory randomization feature of modern OSs.

Using personalized and randomized system call models means that one has to learn the model dynamically for each installation of an application. We address this problem by combining a taint analysis tool with a model learner. One learns some initial model using this learner and the taint analysis makes sure that one does not learn system calls from injected code during this training period. Our current implementation enforces the (initial) model by intercepting system calls with the help of a kernel module.

We need to expect that the initial system call model is not complete and will therefore result in false positives, i.e., some system calls performed by the application are not yet part of the model and will hence result in a model violation. Whenever this happens, we switch to a combined taint and learn mode. The last requests before the model violation are replayed in taint/learn mode. If the taint mode does not detect an intrusion, we update the system call model. Like other approaches that use a replay mechanism (e.g., the RX approach [39], Sweeper [49] or recovery-oriented computing [35]), we need an application-specific proxy to perform this replay. While for stateless (server) applications a proxy is sufficient, for stateful applications, we need periodic application-level checkpoints. Since we do not continuously track taint information in SwitchBlade, we cannot assume that a checkpoint is not already corrupted, i.e., we have the potential for false negatives – unless we mark the complete checkpoint as tainted.

Application-level checkpointing does not write the complete process state but instead writes only the data that is essential for recovery. Marking the complete application-level checkpoint as tainted avoids false negatives and in our experiments also false positives. Note that many modern desktop and server applications support application-level checkpointing. For example, IMAP servers support a CHECK command to write the contents of mailboxes to disk. One can roll back to such an application checkpoint using a process in an uncorrupted initial state. This permits us to roll back to a point in which the taint analysis was switched off and for which we do not have any taint information.

SwitchBlade has several restrictions. First, it is currently only applicable to stateless applications and stateful applications for which an application-level checkpointing mechanism exists. Second, we require the support for replaying application-level requests with the help of a proxy. Replay mechanisms are technically difficult but can be reused to achieve other objectives like locating bugs [48], tolerating software bugs [35, 39, 49], tolerating server crashes and facilitating load-balancing across machines. Third, our implementation has several shortcomings that we are currently working on, in particular, supporting multithreaded applications and kernel support for deterministic replay of system calls. Fourth, we can only detect code injections that modify the system call behavior. However, using randomization we can make it increasingly difficult for injected code to not modify the system call behavior.

In Sec. 2, we review the related work before we present the SwitchBlade architecture in Sec. 3. In Sec. 4, we introduce our system call model. We demonstrate the need for dynamic model updates and we describe our novel dataflow-based learner in Sec. 5. We describe the taint analysis in Sec. 6 and the model enforcement in Sec. 7. The performance and effectiveness of SwitchBlade is investigated in Sec. 8. We believe that our evaluation shows the SwitchBlade can provide an effective and efficient mean for detecting and containing code injection attacks.

2. RELATED WORK

There exist a variety of run-time tools that address certain classes of vulnerabilities, e.g., PointGuard [13], FormatGuard [12], StackGuard [11], Stack Shield, LibSafe [9], ProPolice [18], and LibSafePlus [2] to name a few. Some of these tools require the recompilation of the source code and might therefore not be applicable to protect third-party software. There exist more generic tools that cover a larger class of vulnerabilities, e.g., program shepherding [25], XFI [17], and dynamic taint-based tools like the ones described in [15, 23, 34, 47]. For several of the run-time tools, researchers found ways to circumvent them (e.g., [27, 43]).

There exists a significant body of related work in the domain of system call interception and intrusion detection.
System call interception to confine the intrusion into applications has been used for many years [22, 36, 38]. Several difficulties of ensuring correct system call interception, like dealing with time of check versus time of use problems, have been described in [21] and more recently in [52]. Intercepting security related operations is now supported by default in the Linux kernel [54] and is used by security tools like AppArmor [4, 14].

System call based intrusion detection systems can broadly be classified into misuse-based and anomaly-based systems. Misuse-based systems detect deviations from a usage model while anomaly-based ones detect statistical deviation from safe system call behaviors. There is a rich set of articles about anomaly-based intrusion detection, e.g., [19, 24, 51, 44, 41, 26, 30]. The basic underlying idea is to look at a window of system calls to detect deviations to known good windows of system calls. Newer approaches also use system call arguments for the detection [26] and the dataflow between the system calls [6]. Such systems are susceptible to mimicry attacks that imitate the statistical system call behavior of the application [50].

Misuse-based detection schemes provide a set of rules that describe which system calls are permitted and which are not [22, 38]. SwitchBlade uses a misuse-based detection scheme and applies several mechanisms to prevent exploits from evading detection (see Sec. 4). Creating a good policy with a low false positive and false negative rate is in general difficult. [38] points out the difficulties of generating a good policy for system call interception: (1) during the learning phase one needs to cover all possible code paths to reduce the number of false positives at deploy time, and (2) one needs to make sure to avoid anomalies (like exploits) during the training phase to reduce the number of false negatives. We address these two main issues in SwitchBlade by combining a novel dataflow-oriented model learner and a dynamic taint analysis tool. The authors of [38] also point out that most of the policy violations they experienced were by a web server caused by the attempted execution of user-created CGI scripts. We therefore applied SwitchBlade to two web servers.

The policy used by SwitchBlade is quite different from those used by tools like SysTrace [38] or AppArmor. SwitchBlade focuses on the checking of the sequencing of system calls. The system call model is derived from the model carrying code approach [42] in which the model was used to confine mobile code. We added explicit garbage collection for model variables and we can tag arguments as being constant (see Sec. 4). Also, we learn the system call model via data flow analysis to ensure the accuracy of the dataflow constraints of the models.

Dynamic taint analysis [47, 15, 34, 23, 40] keeps track of untrusted input data and detects attempts to misuse such tainted data, e.g., as a jump target. There exist taint analysis approaches based on emulation using emulators like QEMU [5], dynamic binary rewriting tools like Valgrind [32], or hardware support. Using MMU support, [23] can dynamically switch between executing in QEMU while needing to track tainted data and executing natively when no tainted data is accessed. Recall that SwitchBlade only switches to taint mode after we detected a system call model violation. When a system call model becomes sufficiently complete, this should mainly happen because of the activation of an exploit. SwitchBlade provides in this case a way to reproduce the exploit and hence, a way to help locating and fixing the vulnerability.

One possible application domain of SwitchBlade is to protect applications from worms. A reactive worm defense [10, 8, 37, 49, 7] consists of a set of monitoring sites that detect exploits and generated exploit-specific signatures. These signatures might however not protect against polymorphic worms. One could address polymorphic worms by shipping vulnerability-specific execution filters like [33, 49]: the taint analysis is restricted to known exploits and in this way the overhead of the taint analysis can be reduced. However, all reactive worm defense systems have a vulnerability window that stretches from the time the worm starts spreading until the time a signature or execution filter is received and installed by a node. Sweeper [49] provides additionally a repair mechanism for detected infections by rolling back to an uninfected checkpoint. SwitchBlade would instead permit preventive worm defense. Worms need to communicate with the external world and hence, are forced to perform system calls. This makes them detectable by system call interception and in this way they can be prevented from spreading.

### 3. SWITCHBLADE ARCHITECTURE

The objective of SwitchBlade is to detect exploits with a very high likelihood while ensuring a low false positive rate and a low performance overhead. SwitchBlade’s general architecture is depicted in Fig. 1. SwitchBlade’s primary method for detecting exploits is a process-specific, fine-granular system call model (see Sec. 4). The system call model describes the set of permissible sequences of system calls, provides constraints on the dataflow of arguments of the system calls and the locations they can be called from. In our current implementation, the system call model is enforced with the help of a Linux kernel module.

SwitchBlade requires that we can roll back an application to a previous state in case of a violation of the system call model. We then replay the outstanding requests in taint mode to decide if the model violation was caused by an incomplete model or by an intrusion. In a stateless server,
the outstanding requests are all requests that have not yet been replied to. In a stateful server, these are all requests that need to be processed after the most recent checkpoint was taken.

Because of non-determinism in the execution, a model violation in normal mode might not reoccur during replay. This could result in a performance penalty because we might have to switch multiple times to taint mode and then switch back to normal before, eventually, we might succeed to extend the system call model. In our experience, this is not an issue and models grow nicely with the number of model violations (see also Sec. 8).

Roll back can in general be supported via checkpointing and logging. For logging, we use the standard approach of sending all client interactions through a proxy. The proxy is application-specific and maintains the set of requests since the last checkpoint. Checkpoints contain sufficient state to permit an application to roll back and then replay/continue the execution. In SwitchBlade, we cannot use checkpoints that simply save the content of the entire address space. When restarting the application in taint mode, we would not know which parts of the address space are tainted and which are not. Tainting the entire address space would result in a high false positive rate. Our approach is to use an application-level checkpointing mechanism. Most applications can save the state that is essential to recover from a crash and taint-based approaches can deal with such application-level checkpoints. For example, text editors (like Vim) save the state of the currently edited buffer periodically to disk. For stateless servers, we do not need an application-level checkpointing mechanism. For switching to taint mode, the server is always restarted from its initial state. The proxy then resends all outstanding requests that have not been replied before the model violation.

A more detailed view of SwitchBlade is given in Fig. 2. A server process initially starts in taint mode, forks a child process that switches on system call enforcement for itself before it goes to normal mode to process the first request. Requests are processed until a violation is detected. The parent is notified via a signal of such a violation and spawns a new child process that processes the outstanding requests in taint mode after switching back to normal mode. If during taint mode a security violation is detected (e.g., by using a tainted address as a jump target), all outstanding requests are dropped and the child terminates. If no vulnerability is found in taint mode but a deviation from the current system call model is detected, the system call model is extended to cover the replayed execution. The proxy ensures that the replay is not visible to the client (except for some slight delay). The model is updated in the kernel and system call model enforcement is switched on before the process leaves taint mode.

4. SYSTEM CALL MODEL

To give an intuition for how a system call model looks like, Fig. 3 shows the system call model of the Arithmetic Unixbench benchmark. State l0 is the backtrace of the first getrusage system call. The backtrace of a system call is the sequence of all return addresses on the stack at the time when the system call is issued. The next allowed system call is the second getrusage at node l1 and so on. State l6 is the end state and is not associated with a backtrace. The dataflow between the system calls is constrained: the first argument of the write and the close call have to be the return value of the open call. The digit 4 identifies the register that carries the first system call argument.

More precisely, a system call model is a graph in which each node represents an unique stack backtrace and edges represent system calls with optional argument constraints.

A backtrace is the sequence of return addresses on the stack. One can compute this sequence at runtime (with techniques also used by debuggers) for instance with the help of frame pointers. An edge from a node n1 to node n2 that is labeled “sys_N” means that the system call with system call number N is called with the backtrace that is represented by n1. Because an edge defines the transition from one node to its successor, we use the terms edge and transition synonymously. For any system call number N and any two nodes, there is at most one edge that is labeled with “sys_N”. For better readability we often replace “sys_N” by the name of the system call, e.g., “open” or “write”. Typically, all edges starting from a node refer to the same system call number. However, some library functions do not properly update the frame pointer and hence, some return addresses on the stack are not part of the computed backtrace. This can result in (rare) cases in models in which different system calls are issued with the same backtrace. Also, this means that there can be multiple edges between two nodes.

The outgoing edges of a node n1 determine the set of system calls that are permitted to be executed with the backtrace associated with n1. If there are k outgoing edges from a node n1, then a system call executed with the backtrace represented by n1 has to match at least one of the k edges. Matching means that the system call number N of the edge and the number of the executed system call are identical and all argument constraints are satisfied (see below). The edges that match determine the set of backtraces of the next system call. As soon as the backtrace of the next system call is known, only one edge is permitted to match. Say, that the backtrace of the next system call is represented by node nd, then there must be exactly one edge between nd and n1 that matches the system call with backtrace n1.

The system call model not only determines the permissible sequence of system calls but also restricts the dataflow between system calls. Edges may therefore constrain the arguments of the system calls. We achieve this by introducing variables and constants in the system call model. Each model variable represents a set of values. A model can contain argument constraints of the form ai = vj on edges. This means that the value of the i-th argument (ai) must be in set vj.

Figure 3: Syscall Model of Arithmetic Unixbench Benchmark.
that are passed in registers. Note, the automatically generated model graphs (e.g., see Fig. 3) contain an index to a block containing the value of all registers, i.e., instead of say $a_1 = v_1$ we depict a constraint as $4 = v_1$.

Some arguments are always constant, e.g., one passes always the same filename that is stored at a fixed address and with a constant set of permission flags. We support constraints that say that an arguments is constant. These are of the form $a_i = c$, e.g., $a_1 = 0x3$. One could support a set of constants (as we support variables that contain a set of values; see below). For efficiency, our model and implementation is currently restricted to one value. Fig. 4 depicts a model that contains several constant arguments. Note that if a constant pointer points to a code page or a read-only page, an attacker will not be able to control this argument: this would require to make the page writable which would in turn require a system call which would with a high likelihood not be covered by the system call model. We omitted the constant arguments in all following system call models for readability.

During runtime one can add and remove values from the model variables. The model permits to define one model variable per node. The return value of the system call executed at this node are stored in the set represented by this variable. In the graph, we denote this by edge labels of the form "$v_j = \text{sys\_N}". For example, an open call returns a file descriptor, which might be added to a model variable $v_j$, and a later write call might contain an argument constraint that says the first argument has to be in $v_j$. If a variable is not used as an input argument, we drop it from the model. This means that we do not need to maintain a model variable for each system call during runtime but only for those system calls whose return value may be used as an argument in some later system call.

We have to be able to remove values from the variable sets. To do so, we can add a free attribute to the edge at which a variable value is used last the last time. For example, in Fig. 3 we remove the value from variable $v_{17}$ that is passed as argument to close. However, handles might not always be closed as depicted in Fig. 4, where, no call to close is issued for $v_{20}$. We can nevertheless remove the values from the variable set because from a model perspective, we are only interested in keeping values that might still be used in an argument constraint (see edge from node $l_{21}$ to $l_{23}$).

Our system call model is similar to the one used in [42] to predict what system calls an application will perform. The main differences are with respect to the constraints of arguments in which we restrict ourselves to simple checks on system call arguments passed in registers. Moreover, our model explicitly states when certain model variables can be garbage collected and which arguments are constant. More importantly, we personalize and randomize the model since our use of the model is different from that of [42].

4.1 Personalization

Most injected code will have to perform system calls to achieve its purpose. For example, a worm will try to connect to other hosts, or an injected keyboard logger would need to write the keystrokes somewhere. To stay undetected, an exploit would need to (1) perform system calls in an order that is consistent with the system call model, (2) for each system call, the backtrace needs to match the sequence in

![Figure 2: Execution of an application under SwitchBlade. Violations of the model result in the re-execution of the affected request in taint mode. If the violation was caused by a false positive, the model is updated appropriately.](image)

![Figure 4: Programs might not always close handles. The system call model can sometimes remove values from variable sets even if they are not freed by the program. In this case, the file descriptor is removed from set $r_{20}$ on the last write call.](image)
the model, and (3) the arguments of a system call need to satisfy all argument constraints active for the call.

To maximize the chances that an exploit violates the system call model – and unlike other system call interception approaches – we want to personalize the system call model. Our goal is to use a minimal system call model that describes the permissible sequences of system calls of an application within a given environment. We show in Sec. 5 that system call models are dependent on many factors like the clients of a protected server application and the error rate of the underlying system. Hence, by only permitting the system call sequences SwitchBlade observed in a given environment, we restrict the possibilities of an attacker.

An attacker cannot just study the source code or program traces of another installation to know for sure which sequences are permitted and which are not. For example, in a given environment, certain program features might be disabled by a configuration file. Our system call model should therefore prohibit system call behaviors specific to the disabled features. However, an attacker could try to restrict herself to a minimal system call model. We show in the next section how to minimize the chance of a successful attack by using randomization.

The potential disadvantages of personalizing the system call model are (1) one needs to learn the system call model in each environment, and (2) the probability of false positives will most likely increase because one cannot invest as much time in training in each environment. We address these two issues by automatic learning and updating the system call model during deployment (see Sec. 5). This facilitates the use of personalized system call models without increasing the risk of false positives and with a low training overhead.

4.2 Randomization

Our goal is to detect code injections with a high likelihood even if these do not change the original system call behavior. Personalization makes it already more difficult to find permissible system call sequences. However, an attacker might still find system call sequences that are permitted in most environments. To minimize the chance that an exploit succeeds in imitating the system call behavior even further, we randomize the model in two ways. By default, Linux randomizes the location at which dynamic link libraries are loaded. This automatically randomizes the return addresses in backtraces. Our approach supports this randomization by updating the expected addresses accordingly. An attacker would have to figure out the expected return addresses on the stack for each system call that it wants to perform (without performing other system calls).

We do not solely rely on address space randomization. We also randomize the system call model by injecting random system calls. We implemented a wrapper generator that can wrap library functions used by an application and/or functions within an application. These function wrappers call system calls with a random invalid system call number (32-bit number) before and/or after calling the original function. Fig. 5.b shows a randomized version of the partial Apache model depicted in Fig. 5.a. The execution of an invalid system call is particularly fast because the kernel performs a simple check if the system call number is valid and if it is not, the call just returns with an error code.

An attacker would have to inject code that can guess or somehow figures out at run-time the permitted sequence of

system calls together with the correct backtrace for each system call. One possible way to achieve that would be to emulate the original code using an emulation framework that does neither use library/application functions nor system calls. The original code would be used as an oracle for the right calling sequence for the attack code. To prevent this attack, we want to set all pages containing wrapper code as execution only, i.e., we would prohibit any read accesses of the wrapper code. This would prevent the emulation of the wrappers. Changing the protection of wrapper pages would require a system call that would, by definition, violate the system call model. Supporting execute only pages on IA32 is possible but non-trivial and would require changes to the Linux kernel. So far, we have however not implemented these changes.

Preventing read access to the (wrapper) code, will prevent the emulation of code that might contain system calls. Since an attacker would not know which functions contain system calls and which not, injected code would have to call all functions in the original order to stay undetected. Moreover, changing the arguments to these functions might result in (1) a sequence of system calls that was not experienced during learning or (2) a dataflow between system calls that might not be covered by the model. In other words, by inserting more random invalid system calls in the program code, we can make it more and more difficult for an attacker to come up with a permissible sequence of system calls.

5. MODEL LEARNER

SwitchBlade can learn and update the system call model at runtime. The need for dynamic model updates not only stems from our approach to use personalized and randomized system call models but is a general problem of system call interpolation frameworks. To motivate this, we first show several measurements indicating the problem of coming up with a good system call model, i.e., one that has a low false positive rate.

5.1 Problem: False Positives

The potential disadvantage of model-based system call monitoring is an unacceptably high false positive rate. The model is typically learned by the use of traces (e.g., see [42]). One can easily see that tracing can lead to incomplete models in which nodes (i.e., call locations) and transitions (system calls with argument constraints) are missing. For example, it is unlikely that all error handling code is executed during tracing. Typically, error handlers will issue addi-
Figure 6: Model size of grep grows when we inject errors in the form of returning error codes for random system calls during tracing. The number of states grows from 36 to 79 and the number of transitions from 46 to 129.

Figure 7: Model sizes for Apache when learned with different client programs.

Figure 8: Overlap of the Apache system call models generated with 5 different client programs: left graph shows overlap in nodes and the right the overlap in edges.

Figure 9: Transitions and edges missing in the wget Apache model (dotted lines) in comparison to the unified Apache model.

5.2 Dataflow-Based Learner

Sekar et al. [42] describe a trace-based learner that tries to match values returned by some system call with arguments passed into subsequent system calls. For example, if there are two calls to open and the first returns value 1 and the second 2 and a later call to write uses value 2 for the file descriptor, the system learns that the 2nd open call and the write call are linked. The trace-based learner needs a specification that relates return values and arguments of different system calls (e.g., the return value of open can be used as argument for read, write and close). Learning in this way can result in wrong connections between system calls because values can be identical for other reasons. Also, some system calls $S_1, ..., S_k$ in a trace might return the same value $x$ and $x$ is later passed as an argument to another system call $S_p$ and it is not always clear which (if any) system call $S_p$ ($p \in \{1, ..., k\}$) did indeed produce value $x$. In other clients (e.g.: User-Agent field) and different connecting handling.

A major problem of the enforcement of an incomplete system call model is that it might be used for denial of service attacks. For Apache, it might be sufficient for an attacker to browse with an uncommon browser to cause the abort of an Apache worker process. While Apache will spawn a new worker process, this might nevertheless lead to a major reduction in throughput for all clients when a new browser version becomes available. For example, even minor changes in browser version (like from Firefox 1.5 to Firefox 1.5.10; see Fig. 7) can modify the system call model. This can lead to model violations which in turn can result in an increased service unavailability.
words, it is not always certain which system calls should be connected via argument constraints in the system call model. To distinguish between the different options, one would have to try to produce additional traces in which the system calls return different values that permit the learner to decide which system call produced a certain argument.

SwitchBlade tracks the dataflow of an application to learn the system call model of an application. In the above example, the dataflow analysis permits us to determine exactly which (if any) of the system calls produced the value $x$. Also, the dataflow analysis permits us to determine if some of the arguments might be constants.

To track the dataflow between system calls, SwitchBlade dynamically instruments the binary code using Valgrind [32], a binary instrumentation and analysis framework. The learning of the model is always performed in taint mode, i.e., in a mode in which we anyhow track the dataflow of an application. Hence, we use a combined Valgrind tool for taint analysis (see Sec. 6) and learning.

Our instrumentation assigns each executed system call an unique id (SCID) and records the backtrace of this call. Each return value of a system call is tagged with the SCID. Whenever this value is copied to another memory location or register, the new location/register is also tagged with the SCID. If a location is cleared or modified in some other way, we remove the SCID from this location. We use the standard approach of keeping a shadow memory [31] to track the SCID; we maintain a 32-bit of shadow information for each register and potentially each word in memory. The shadow information does not only contain the SCID but also whether the content of the location contains a constant and whether it is tainted (see Sec. 6).

To track if a word contains a constant, we initially mark all code segments as containing constants. Whenever a word is read from a code page, the destination is also marked as constant. However, even if two constants are used as an input of an operation, we do not mark the result as a constant. Note that even if a memory location holding a program variable $v$ is marked as constant, this does not necessarily mean that $v$ is indeed a constant: the value of $v$ can of course depend on the control flow of a program. We use this marking to determine if an argument of a system call could be marked as constant in the system call model. This reduces the overhead of the learner because we only attempt to determine if an argument is a constant if it is marked as such by the dataflow analysis.

For each system call $S_i$, we record the system call number $SN$, the backtrace $b$, the SCIDs of the arguments, and if arguments are marked as being constant. If the backtrace $b$ is not yet part of the current model, we add a new node $N_b$ that denotes the new backtrace. When executing the next system call $S_{i+1}$ with a backtrace $B$, we check if there is already an edge between $N_b$ and $N_B$ that is marked with $SN$. If this edge does not yet exist, we add an edge that is marked with $SN$ and argument constraints. For each argument $a_i$ that contains a SCID, we add an argument constraint $a_i = v_x$, where, $v_x$ is the model variable associated with the SCID. If such a model variable does not yet exist, we add it to the model. For example, if the SCID refers to a system call with a backtrace $x$, we add to all outgoing edges from node $N_x$ the prefix “$v_x = ...$”. If an argument $a_i$ is marked as a constant, we add a constraint $a_i = c$, where, $c$ is the value that was passed as argument.

<table>
<thead>
<tr>
<th>Memory to check</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>tainted return address</td>
<td>x</td>
</tr>
<tr>
<td>tainted jump addresses calculated at runtime</td>
<td>x</td>
</tr>
<tr>
<td>tainted format strings</td>
<td>x</td>
</tr>
<tr>
<td>tainted code at jump targets</td>
<td>x</td>
</tr>
<tr>
<td>tainted heap meta data before free</td>
<td>x</td>
</tr>
</tbody>
</table>

Table 1: Checks done by TaintCheck to detect and block exploits.

If the edge already exists, we check if it is consistent with our current observation. It is of course possible that the argument constraints of an edge might change between different calls. For example, the first time the edge is executed, argument $a_i$ may contain a value of some $v_B$ while in the second call it may contain the value of a different model variable $v_B$. To address this issue, we could permit disjunctions in constraints, i.e., $a_i = v_B \lor v_B$. For now, we decided to keep the model simple and instead remove a constraint if we observe conflicting constraints.

Our learner does not only learn data flow that can be derived from the system call specification (e.g.: the file descriptor argument of read was returned by some open). Other data flow like using the returned memory address from a mmap call as buffer for a recv is also learned.

6. Taint Analysis

Taint analysis is a well-known approach to detect and block code exploits. It can be implemented using a dynamic binary instrumentation framework like Valgrind [32]. One of the main advantages of this approach is that it works for arbitrary binaries without the need for recompilation. We have reimplemented the taint analysis tool TaintCheck [34] using our own dataflow engine. We added the support to generate traces to learn system call models (see Sec. 5). We also modified Valgrind to be able to switch to a native execution to remove the overhead of Valgrind while running with system call enforcement switched on.

6.1 TaintCheck

For completeness, we briefly describe TaintCheck. Valgrind runs an application on a simulated CPU under the control of our TaintCheck tool. In this way, TaintCheck marks data from suspicious sources as tainted, traces tainted data flow and blocks the usage of tainted data at vulnerable points. TaintCheck has a very low false positive and false negative rate. One reason why tools like the original TaintCheck are not more widely used to protect vulnerable applications is Valgrind’s enormous slowdown (see Sec. 8).

TaintCheck maintains for each byte of memory and all CPU registers a shadow bit. This shadow bit is set to 1 if the corresponding byte of memory or register is tainted. Initially, all shadow bits are set to 0. In our current configuration, all data that is read form the network is marked as tainted. Optionally, data from the file system can be marked as tainted too. Additionally, the software itself can mark data as tainted. Whenever memory words or registers are copied, their corresponding shadow bits are copied too. In that way, we trace the propagation of tainted data throughout the address space of the application.

Tab. 1 lists all checks executed by TaintCheck to detect and block exploits. All data is checked before it is used as a jump target, as format strings or to concatenate heap blocks.
after freeing them. Whenever a taint check evaluates to true, the corresponding operation is not executed. Instead, the current process is aborted with a detailed error message. The last two checks in Tab. 1 are not part of the original TaintCheck. We added them for completeness and better debugging. The first check is necessary, if an attacker is able to overwrite code without changing the control flow. The second check is redundant because it should not detect anything that would not be detected by the other checks. However, it directly points to the vulnerable heap block.

6.2 Escaping Valgrind
Initially, we start the application under the control of TaintCheck. For stateless server applications, we take one process-level checkpoint (which is currently implemented by a simple fork) to be able to restart quickly in taint mode in case the model enforcer detects a model violation. After taking the checkpoint, we switch on the model enforcement and then switch from the simulated CPU under the control of Valgrind and TaintCheck to the real CPU. Therefore, we implemented an Escape feature for Valgrind. When an application wants to escape Valgrind, the state of the virtual CPU is copied to the real one and the application’s execution then proceeds on the real CPU. This is possible because Valgrind does not modify the memory layout of the application running on the simulated CPU.

TaintCheck stops after the escape. After that, no data is marked tainted and no data flow is traced. While this might permit an attacker to change the control flow without being detected and to contaminate the application’s state, the model enforcer confines the effects of such an attack and will detect it as soon as the system call behavior deviates from the model. Our current implementation does not permit to switch back to Valgrind after an escape. This would be difficult to do correctly: if a vulnerability is detected by the model enforcer, the application’s state and the control flow might have already been contaminated by an attacker. Even if one could roll back to, say, some earlier process-level checkpoint, we would neither have taint information (unless we roll back to a point before we escaped from Valgrind) nor would we know if the state might be corrupted already. We address this issue by using application-level checkpoints to which we roll back for replay.

6.3 Replay of Requests
After the model enforcer detects an attack, it stops the application. To test if this was a false positive because of an incomplete model, the outstanding requests are re-executed (see Fig. 2). Because the current state of the application might be contaminated by an attacker, we roll back to a known clean state that was still under the control of TaintCheck.

If the application is stateful, we load the most recent application-level checkpoint from disk. As this checkpoint could be already contaminated by an attacker the complete checkpoint is marked as tainted, i.e., we do not trust the information that is in the checkpoint. From here on, we replay all outstanding requests by asking the proxy to transmit these again.

During replay, the execution does not escape from TaintCheck. TaintCheck verifies during the re-execution of the outstanding requests that no attack is taking place. The Model Learner checks if the model needs to be updated. Hence, in taint mode the enforcement of the system call model is switched off. If no exploit is detected during the re-execution, the learner will update the model.

The proxy ensures that a client does not see that its requests are replayed, i.e., duplicate replies are filtered by the proxy. SwitchBlade is transparent for client applications.

For simple stateless server applications, we created a simple wrapper library that provide transparent support for running under SwitchBlade. In particular, we do not require any changes of the source code for stateless server applications that serve a single connection per execution (e.g., servers spawned via inetd). The generic wrapper library takes the initial process-level checkpoint (e.g., using fork) and switches from TaintCheck to model enforcing. We have used this approach for the micro benchmarks in Sec. 8.4. Servers serving multiple connections need some modifications: some code for the initial checkpointing needs to be added to the application.

7. MODEL ENFORCEMENT
The Model Enforcement ensures that an application escaped from the control of TaintCheck follows its system call model. Therefore, it must intercept all system calls the application performs, compute the system calls backtrace and compare them to the possible next states of the system call model.

We implemented our model enforcement tool as kernel module for Linux 2.6.20. Using the Linux Security Module (LSM) which is an interface to intercept security-related operations [54] would have been one option. Mandatory Access Control (MAC) systems like AppArmor and SELinux are implemented on top of it. Our system call models are more fine-grained than the security policies of AppArmor and SELinux. We believe that SwitchBlade might therefore detect attacks earlier because an attacker might have to perform some system calls to setup the system call that is detected by the MAC system. We did not use LSM to intercept the systems calls because LSM does not intercept all system calls. Instead we use the utrace framework [29].

When a process switches from TaintCheck to model enforcement, TaintCheck informs the model enforcer that the current process needs to be enforced. Directly after this, the control is switched from the simulated CPU to the real CPU. This switch does not need a system call and therefore, it does not appear in the system call model. The next system call performed by the application has to be from one of the starting states of the system call model.

When a process wants the system call model to be enforced, the model enforcer allocates space for the model and the model variables and sets the starting state of the enforced model. The model enforcer intercepts all system calls from all processes. When a system call is performed by an enforced process, the associated model is checked before performing the system call. Otherwise, the call is forwarded without further checking. The loading of models and request for enforcement are communicated via system calls. All communication attempts by an already enforced process are rejected by the model enforcer. So, there is no possibility for an enforced process to communicate to the model enforcer to ensure that an enforced process cannot break out of the model enforcer’s control.

Before the current system call of an enforced process can be checked, the backtrace of the system call is extracted from
Native SwitchBlade

<table>
<thead>
<tr>
<th>Attacks</th>
<th>Native</th>
<th>SwitchBlade</th>
</tr>
</thead>
<tbody>
<tr>
<td>succeeded</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>failed</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>detected &amp; aborted</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>crashed</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 2: Result of Wilander's testbed running natively and under SwitchBlade.

8. EVALUATION

In our evaluation of SwitchBlade we wanted to test in how far we have achieved our three goals: low false positive rate, low false negative rate and good performance. Therefore, we ran a series of microbenchmarks and applied our approach to the Apache web server. As expected, we experienced a number of false positives during system call model enforcement. However, we did not observe any false positives during replay in taint mode in any of our runs.

All experiments were performed in a virtual machine with 512 MB running on an Intel Core 2 Duo with 2 GBes RAM. We used Ubuntu 7.04 as operating system.

8.1 Synthetic Exploits

The Wilander Testbed [53] is a set of 18 vulnerabilities and exploits. In our environment and without SwitchBlade, of the 18 exploits only six were successful, four resulted in crashes and 8 had no effect. SwitchBlade detects and confines all six of the natively successful exploits.

We used the Wilanders testbed with our generic check pointing wrapper (see Sec. 6.3). We generated an initial system call model for the run of the testbed by printing the help messages – the only model in which no attack is attempted. This model consists of 27 states and 26 transitions. All possible next states are computed. If necessary, the system call’s return value is recorded in the correct model variable. Also, if requested by the model, values are removed from model variables.

If a process violates the current model, the system call is not performed. Instead, the process is stopped but not immediately killed. The parent process stays in taint mode and waits for the child process to terminate or being stopped. The parent process is informed, when the child is stopped. Now it kills the child and forks a new child. This child will stay in taint mode to replay the last requests.

The model enforcer can deal with address space randomization. When a process wants to be enforced, the model enforcer extracts the mapping of the dynamic linked libraries to address ranges. This facilitates the comparison of the backtraces of the model (in which an address is represented as an offset and an unique id of a dynamic linked library) and the backtraces extracted from the stack (in which addresses are converted to offsets and unique ids of dynamic linked libraries).

8.2 Apache

We applied SwitchBlade to the Apache Webserver. Our experiences and the performance we measured are as follows. First, we inserted the code for check-pointing and for replaying outstanding requests. We added 181 lines to server/mpm/prefork/prefork.c. Originally, this file alone contained 1478 lines. The changes within the file were very localized. The effort of adding custom check-pointing and replay support to Apache was therefore very small. We expect a similar effort for other server applications (see Sec. 8.3 for GazTek HTTP Daemon). Apache supports loading plugins (called Apache modules) at run time. SwitchBlade does not require a modification or recompilation of any Apache modules. Apache uses several worker processes to serve concurrent requests. SwitchBlade protects each worker separately.

Fig. 10 shows the overhead of the average connection time for different kinds of content. In all three configurations we used a proxy between the client and the Apache server. The overhead of SwitchBlade in normal mode (i.e., system call enforcement) and executing the same requests in SwitchBlade’s taint mode compared to a native execution without SwitchBlade. We measured the average connection time as seen by a client connecting to the proxy for different kinds of content.

We measured how the Apache system call model evolves with an increasing number of requests (see Fig. 11). For each requests that requires a new state or transition, a new model needs to be generated. Initially, there are many new models but after about a hundred requests the model starts to stabilize. In other words, even without a learning phase,
one can expect that the number of model changes and hence, switches to taint mode, will typically happen less and less often. However, as we pointed out several times, we cannot ever be sure that we reached the maximum model of a certain installation.

8.3 Exploits

We have tested SwitchBlade against three real life exploits: two for the PHP module for Apache \([46, 45]\), and one for the GazTek HTTP Daemon v1.4-3 (ghttpd) \([1]\). All three exploits were detected and aborted. Tab. 3 summarizes our experiments. We tested two exploits for the same vulnerability in the PHP engine. One exploit injected code \([46]\) without touching the control flow and the other one changed the control flow \([45]\). All three exploits violated the system call model and forced the server to replay the request in taint mode. The taint analysis was able to detect and block all three exploits.

For ghttpd, we had to add some code for the check-pointing and replay. We added 133 lines to main.c which originally contained 235 lines.

8.4 Microbenchmarks

We implemented several micro benchmarks to compare the performance overhead of the system call model enforcement and the taint analysis for common command line tools. We used our generic wrapper library (see Sec. 6.3). Hence, the command line tools did not need to be changed.

As expected, the system call models were different for different input files. We first used small input files (40 KBytes) to generate models for the five utilities and second, we used much larger input files (5.7 MBytes). We observed that the system call models of grep, diff and ruby (which runs the same script independent of the input file) differs for the small and the large input files (see Fig. 13). This is another indication that generating a general system call model for an application can be difficult.

Fig. 12 compares the overhead of system call model enforcement to the overhead of taint analysis for four tools. We exclude the runtime measurements for grep because its execution time – even for large input files – was too brief to get a reproducible measurement error. We measured the runtime of the tools starting at the time when the system switches from taint mode to model enforcement. For the taint mode measurements, this switch was disabled. The model enforcement overhead varied between 18% for gzip and 81% for diff. The overhead is larger than that reported for other system call interception tools (e.g., SysTrace \([38]\), AppArmor and SELinux). This has two reasons: (1) our system call model is much more fine-grained. All system calls must be intercepted to minimize the possibilities for the attacker to make an attack conform to the model and to reduce the detection delay. Whereas other system call interception tools are only interested in the security related subset of the system calls. (2) for each system call, the backtrace must be computed. The enormous overhead of the taint analysis is due to we did not optimize it for performance as we use it only as fall-back. Additionally, taint analysis does also track the data flow of system call return values and constants.

8.5 Model Size

The model generated in \([42]\) used the first return address on the stack that points within the application to identify a system call. Our backtrace also uses all return addresses in between to identify a node. In that way, our model becomes more fine grained. For example, the model contains parts associated to individual calls to dynamically linked libraries that issue system calls. For instance, our model of a call to the library function gethostbyname contains 26 distinct states with 28 transitions. Whereas, a model that only uses the return addresses within the application has only one state where all transitions are attached to. An attacker
<table>
<thead>
<tr>
<th>Application</th>
<th>Vulnerability</th>
<th>Exploit</th>
<th>Aborted?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache + PHP 4.4.4</td>
<td>double free in garbage collector</td>
<td>inject code</td>
<td>yes</td>
</tr>
<tr>
<td>Apache + PHP 4.4.4</td>
<td>double free in garbage collector</td>
<td>changes control flow</td>
<td>yes</td>
</tr>
<tr>
<td>GazTek HTTP v1.4-3</td>
<td>stack overflow in function Log</td>
<td>changes control flow</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 3: Results of testing SwitchBlade against real world exploits.

![Figure 14: Size of Apache model depending on the length of the backtrace.](image)

Figure 14: Size of Apache model depending on the length of the backtrace.

![Figure 15: Growing size of the system call model of Vim.](image)

Figure 15: Growing size of the system call model of Vim.

gaining access to the library implementing gethostbyname can issue all system calls of the 28 transitions in an arbitrary order without violating the model. To quantify the impact of the number of return addresses used for a state label, we generated different models for the Apache worker process by restricting the backtrace length. Fig. 14 shows the number of states and transitions and the maximum and average outdegree of the generated models. The longer the backtrace, the larger the number of states and transitions. The outdegree of the states shrinks with a growing length of the backtrace. This indicates that models based on longer or unrestricted backtraces will be more restrictive. A more restrictive model will make it more difficult for an attacker to perform system calls without violating the system call model.

8.6 Stateful Application

We have tested SwitchBlade with the text editor Vim. Vim performs periodic application-level checkpoints. Using some initial system call model various actions like saving, calling external applications, and using the online help resulted in model violations from an initially learned model. Our wrapper detects this and restarts Vim under TaintCheck. Vim restores its checkpoint, we can repeat the last action and restart Vim under model enforcement again. The model was extended in each step according to TaintCheck’s trace. Fig. 15 shows the growth of Vim’s system call model. The growing of average out-degree from 1.5 to 1.9 (right Y axis) shows that there were more new control flow operations as new system calls.

9. CONCLUSION

The SwitchBlade system uses a novel combination of system call interception (in normal mode) and dynamic taint analysis (when checking violation of the system call model). The normal mode is efficient and we use randomization of the model to make code injections arbitrarily difficult. The dynamic taint analysis integrates our novel dataflow-based learner to update too strict models. In our experiments, SwitchBlade was able to detect all exploits without any difficulties.

The main obstacle in using SwitchBlade in practice is the need for a replay mechanism (to verify system call model violations), i.e., for checkpointing and logging. However, new dependability approaches like RX [39] use replay mechanisms to increase the availability of applications. A common replay mechanism for dependability and security is possible and hence, one could amortize its cost.

SwitchBlade can cope with both stateless and stateful applications. For stateless approaches, we perform one simple fork-based checkpoint at the start of the execution to speed up the switch from normal mode to taint mode. There is no need for an elaborate checkpointing mechanism for stateless servers. However, stateful applications need to be checkpointed periodically. The difficulty in replaying requests of a stateful application is that we do not have the taint information for the last checkpoint. Hence, we use application-level checkpoints, i.e., one that only stores the essential state needed for restart. Many user and server applications already support such application-level checkpoints. The dynamic taint analysis conservatively marks the complete checkpoint as tainted during restart. In our experiments this did not result in any false positives during replay. An interesting observation is that such application-level checkpoints are also most suited for recovering from application crashes that are caused by software bugs [9].

Acknowledgements

We thank Michael Rudolf and all anonymous reviewers for their efforts to improve the final version of this paper.

10. REFERENCES

runtime protection against buffer overflows. 


[31] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In


