Automating Mimicry Attacks Using Static Binary Analysis

Christopher Kruegel and Engin Kirda
Technical University Vienna
chris@auto.tuwien.ac.at, engin@infosys.tuwien.ac.at

Darren Mutz, William Robertson, and Giovanni Vigna
Reliable Software Group, University of California, Santa Barbara
{dhm,wkr,vigna}@cs.ucsb.edu

Abstract

Intrusion detection systems that monitor sequences of system calls have recently become more sophisticated in defining legitimate application behavior. In particular, additional information, such as the value of the program counter and the configuration of the program's call stack at each system call, has been used to achieve better characterization of program behavior. While there is common agreement that this additional information complicates the task for the attacker, it is less clear to which extent an intruder is constrained.

In this paper, we present a novel technique to evade the extended detection features of state-of-the-art intrusion detection systems and reduce the task of the intruder to a traditional mimicry attack. Given a legitimate sequence of system calls, our technique allows the attacker to execute each system call in the correct execution context by obtaining and relinquishing the control of the application's execution flow through manipulation of code pointers.

We have developed a static analysis tool for Intel x86 binaries that uses symbolic execution to automatically identify instructions that can be used to redirect control flow and to compute the necessary modifications to the environment of the process. We used our tool to successfully exploit three vulnerable programs and evade detection by existing state-of-the-art system call monitors. In addition, we analyzed three real-world applications to verify the general applicability of our techniques.

Keywords: *Binary Analysis, Static Analysis, Symbolic Execution, Intrusion Detection, Evasion.*

1 Introduction

One of the first host-based intrusion detection systems [5] identifies attacks by finding anomalies in the stream of system calls issued by user programs. The technique is based on the analysis of fixed-length sequences of system calls. The model of legitimate program behavior is built by observing normal system call sequences in attack-free application runs. During detection, an alert is raised whenever a monitored program issues a sequence of system calls that is not part of the model.

A problem with this detection approach arises in situations where an attack does not change the sequence of system calls. In particular, the authors of [17] observed that the intrusion detection system can be evaded by carefully crafting an exploit that produces a legitimate sequence of system calls while performing malicious actions. Such attacks were named *mimicry attacks*.

To limit the vulnerability of the intrusion detection system to mimicry attacks, a number of improvements have been proposed [4, 9, 14]. These improvements are based on additional information that is recorded with each system call. One example [14] of additional information is the origin of the system call (i.e., the address of the instruction that invokes the system call). In this case, the intrusion detection system examines the value of the program counter whenever a system call is performed and compares it to a list of legitimate "call sites." The idea was extended in [4] by incorporating into the analysis information about the call stack configuration at the time of a system call invocation.

A call stack describes the current status and a partial history of program execution by analyzing the return addresses that are stored on the program's run-time stack. To extract the return addresses, it is necessary to unwind the stack, frame by frame. Figure 1 shows a number of frames on a program stack and the chain of frame (base) pointer references that are used for stack unwinding.

Checking the program counter and the call stack at each system call invocation serves two purposes for the defender. First, the check makes sure that the system call

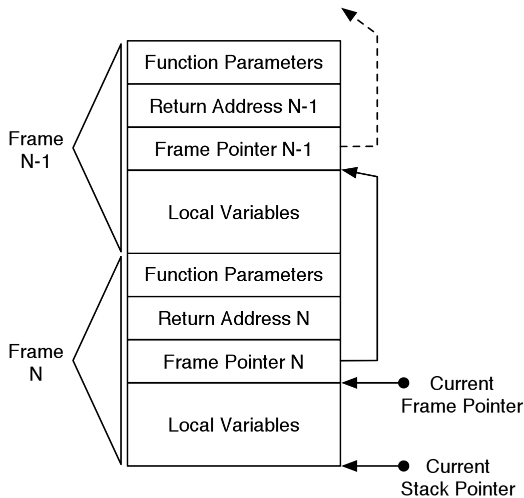

Figure 1: Call stack and chain of frame pointers.

was made by the application code. This thwarts all code injection attacks in which the injected code directly invokes a system call. Second, after a system call has finished, control is guaranteed to return to the original application code. This is because the return addresses on the stack have been previously verified by the intrusion detection system to point to valid instructions in the application code segment. This has an important implication. Even if the attacker can hijack control and force the application to perform a single system call that evades detection, control would return to the original program code after this system call has finished.

The common agreement is that by including additional information into the model, it is significantly more difficult to mount mimicry attacks [3]. However, although additional information undoubtedly complicates the task of the intruder, the extent to which the attack becomes more complicated is less clear. System-call-based intrusion detection systems are not designed to *prevent* attacks (for example, buffer overflows) from occurring. Instead, these systems rely on the assumption that any activity by the attacker appears as an anomaly that can be detected. Unfortunately, using these detection techniques, the attacker is still granted full control of the running process. While the ability to invoke system calls might be significantly limited, arbitrary code can be executed. This includes the possibility to access and modify all writable memory segments.

The ability to modify program variables is in itself a significant threat. Consider, for example, an attacker that alters variables that are subsequently used as `open` or `execv` system call parameters. After the modification, the attacker lets the process continue. Eventually, a system call is invoked that uses values controlled by the attacker. Because the system call is made by legitimate application code, the intrusion remains undetected.

In some cases, however, modifying program variables is not sufficient to compromise a process and the attacker is required to perform system calls. Given the assumption that an attacker has complete knowledge about the detection technique being used, it is relatively straightforward to force the application to perform one undetected system call. To do so, the attacker first pushes the desired system call parameters on the stack and then jumps directly to the address in the application program where the system call is done. Of course, it is also possible to jump to a library function (e.g., `fopen` or `execlp`) that eventually performs the system call. Because it is possible for the injected code to write to the stack segment, one can inject arbitrary stack frames and spoof any desired function call history. Thus, even if the intrusion detection system follows the chain of function return addresses (with the help of the stored base pointers), detection can be evaded.

The problem from the point of view of the attacker is that after the system call finishes, the checked stack is used to determine the return address. Therefore, program execution can only continue at a legitimate program address and execution cannot be diverted to the attacker code. As a consequence, there is an implicit belief that the adversary can at most invoke a single system call. This constitutes a severe limitation for the intruder, since most attacks require multiple system calls to succeed (for example, a call to `setuid` followed by a call to `execve`). This limitation, however, could be overcome if the attacker were able to somehow *regain* control after the first system call completed. In that case, another forged stack can be set up to invoke the next system call. The alternation of invoking system calls and regaining control can be repeated until the desired sequence of system calls (with parameters chosen by the attacker) is executed.

For the purpose of this discussion, we assume that the attacker has found a vulnerability in the victim program that allows the injection of malicious code. In addition, we assume that the attacker has identified a sequence of system calls $s_1, s_2, \ldots, s_n$ that can be invoked after a successful exploit without triggering the intrusion detection system (embedded within this sequence is the attack that the intruder actually wants to execute). Such a sequence could be either extracted from the program model of the intrusion detection system or learned by observing legitimate program executions. By repeatedly forcing the victim application to make a single undetected system call (of a legitimate sequence) and later regaining control, the protection mechanisms offered by additional intrusion detection features (such as checking return addresses or call histories) are circumvented. Thus, the task of the intruder

is reduced to a traditional mimicry attack, where only the order of system calls is of importance.

In this paper, we present techniques to regain control flow by modifying the execution environment (i.e., modifying the content of the data, heap, and/or stack areas) so that the application code is forced to return to the injected attack code at some point after a system call. To this end, we have developed a static analysis tool that performs symbolic execution of x86 binaries to automatically determine instructions that can be exploited to regain control. Upon detection of exploitable instructions, the code necessary to appropriately set up the execution environment is generated. Using our tool, we successfully exploited sample applications protected by the intrusion detection systems presented in [4] and [14], and evaded their detection.

The paper makes the following primary contributions:

- We describe novel attack techniques against two well-known intrusion detection systems [4, 14] that evade the extended detection features and reduce the task of the intruder to a traditional mimicry attack.
- We implemented a tool that allows the automated application of our techniques by statically analyzing the victim binary.
- We present experiments where our tool was used to generate exploits against vulnerable sample programs. In addition, our system was run on real-world applications to demonstrate the practical applicability of our techniques.

Although our main contributions focus on the automated evasion of two specific intrusion detection systems, an important point of our work is to demonstrate that, in general, allowing attackers to execute arbitrary code can have severe security implications.

The paper is structured as follows. In Section 2, we review related work on systems that perform intrusion detection using system calls. In Section 3, we outline our techniques to regain control flow. Section 4 provides an in-depth description of our proposed static analysis and symbolic execution techniques. In Section 5, we demonstrate that our tool can be used to successfully exploit sample programs without raising alarms. In addition, the system is run on three real-world applications to underline the general applicability of our approach. Finally, in Section 6, we briefly conclude and outline future work.

2 Related Work

System calls have been used extensively to characterize the normal behavior of applications. In [7], a classification is presented that divides system-call-based intrusion detection systems into three categories: "black-box", "gray-box", and "white-box". The classification is based on the source of information that is used to build the system call profiles and to monitor the running processes.

Black-box approaches only analyze the system calls invoked by the monitored application without considering any additional information. The system presented in [5], which is based on the analysis of fixed-length *sequences of system calls*, falls into this category. Alternative data models for this approach were presented in [18], while the work in [19] lifted the restriction of fixed-length sequences and proposed the use of variable-length patterns. However, the basic means of detection have remained the same.

Gray-box techniques extend the black-box approaches by including additional run-time information about the process' execution state. This includes the origin of the system call [14] and the call stack [4], as described in the previous section. Another system that uses context information was introduced in [6]. Here, the call stack is used to generate an execution graph that corresponds to the maximal subset of the program control flow graph that one can construct given the observed runs of the program.

White-box techniques extract information directly from the monitored program. Systems in this class perform static analysis of the application's source code or binary image. In [16], legal system call sequences are represented by a state machine that is extracted from the control-flow graph of the application. Although the system is guaranteed to raise no false positives, it is vulnerable to traditional mimicry attacks. Another problem of this system is its run-time overhead, which turns out to be prohibitively high for some programs, reaching several minutes per transaction. This problem was addressed in [8], using several optimizations (e.g., the insertion of "null" system calls), and later in [9], where a Dyck model is used. For this approach, additional system calls need to be inserted, which is implemented via binary rewriting.

An approach similar to the one described in the previous paragraph was introduced in [11]. In this work, system call inlining and "notify" calls are introduced instead of the "null" system calls. Also, source code is analyzed instead of binaries. Another system that uses static analysis to extract an automaton with call stack information was presented in [3]. The work in this paper is based on the gray-box technique introduced in [4]. In [20], waypoints

are inserted into function prologues and epilogues to restrict the types of system calls that they can invoke.

Black-box and gray-box techniques can only identify anomalous program behavior on the basis of the previous execution of attack-free runs. Therefore, it is possible that incomplete training data or imprecise modeling lead to false positives. White-box approaches, on the other hand, extract their models directly from the application code. Thus, assuming that the program does not modify itself, anomalies are a definite indication of an attack. On the downside, white-box techniques often require the analysis of source code, which is impossible in cases where the code is not available. Moreover, an exhaustive static analysis of binaries is often prohibitively complex [6].

This paper introduces attacks against two gray-box systems. Thus, related work on attacking system-call-based detection approaches is relevant. As previously mentioned, mimicry attacks against traditional black-box designs were introduced in [17]. A similar attack is discussed in [15], which is based on modifying the exploit code so that system calls are issued in a legitimate order. In [7], an attack was presented that targets gray-box intrusion detection systems that use program counter and call stack information. This attack is similar to ours in that it is proposed to set up of a fake environment to regain control after the invocation of a system call. The differences with respect to the attack techniques described in this paper are twofold. First, the authors mention only one technique to regain control of the application's execution flow. Second, the process of regaining control is performed completely manually. In fact, although the possibility to regain control flow by having the program overwrite a return address on the stack is discussed, no example is provided that uses this technique. In contrast, this paper demonstrates that attacks of this nature can be successfully automated using static analysis of binary code.

3 Regaining Control Flow

In this section, we discuss possibilities to regain control after the attacker has returned control to the application (e.g., to perform a system call). To regain control, the attacker has the option of preparing the execution environment (i.e., modifying the content of the data, heap, and stack areas) so that the application code is forced to return to the attacker code at some point after the system call. The task can be more formally described as follows: Given a program p, an address s, and an address t, find a configuration C such that, when p is executed starting from address s, control flow will eventually reach the target address t. For our purposes, a configuration C comprises all values that the attacker can modify. This includes the contents of all processor registers and all writable memory regions (in particular, the stack, the heap, and the data segment). However, the attacker cannot tamper with write-protected segments such as code segments or read-only data.

Regaining control flow usually requires that a code pointer is modified appropriately. Two prominent classes of code pointers that an attacker can target are *function pointers* and *stack return addresses*. Other exploitable code pointers include *longjmp buffers*.

A function pointer can be modified directly by code injected by the attacker before control is returned to the application to make a system call. Should the application later use this function pointer, control is returned to the attacker code. This paper focuses on binary code compiled from C source code, hence we analyze where function pointers can appear in such executables. One instance is when the application developer explicitly declares pointer variables to functions at the C language level; whenever a function pointer is used by the application, control can be recovered by changing the pointer variable to contain the address of malicious code. However, although function pointers are a commonly used feature in many C programs, there might not be sufficient instances of such function invocations to successfully perform a complete exploit.

A circumstance in which function pointers are used more frequently is the invocation of shared library functions by dynamically linked ELF (executable and linking format) binaries. When creating dynamically linked executables, a special section (called procedure linkage table – PLT) is created. The PLT is used as an indirect invocation method for calls to globally defined functions. This mechanism allows for the delayed binding of a call to a globally defined function. At a high level, this means that the PLT stores the addresses of shared library functions. Whenever a shared function is invoked, an indirect jump is performed to the corresponding address. This provides the attacker with the opportunity to modify the address of a library call in the PLT to point to attacker code. Thus, whenever a library function call is made, the intruder can regain control.

The redirection of shared library calls is a very effective method of regaining control, especially when one considers the fact that applications usually do not invoke system calls directly. Instead, almost all system calls are invoked through shared library functions. Thus, it is very probable that an application executes a call to a shared function before every system call. However, this technique is only applicable to dynamically linked binaries. For statically

linked binaries, alternative mechanisms to recover control flow must be found.

One such mechanism is the modification of the function return addresses on the stack. Unfortunately (from the point of view of the attacker), these addresses cannot be directly overwritten by the malicious code. The reason, as mentioned previously, is that these addresses are checked at every system call. Thus, it is necessary to force the application to overwrite the return address *after* the attacker has relinquished control (and the first system call has finished). Also, because the stack is analyzed at every system call, no further system calls may be invoked between the time when the return address is modified and the time when this forged address is used in the function epilogue (i.e., by the `ret` instruction).

In principle, every application instruction that writes a data value to memory can be potentially used to modify a function return address. In the Intel x86 instruction set, there is no explicit store instruction. Being based on a CISC architecture, many instructions can specify a memory location as the destination where the result of an operation is stored. The most prominent family of instructions that write data to memory are the data transfer instructions (using the `mov` mnemonic).

int global;
 	
void f()
{
  global = 0;
}
	
	movl $0x0,0x8049578

(a) Direct variable access

int global;
 	
void f()
{
  int *p = &global;
 	
  *p = 0;
}
	
	movl $0x8049578,0xfffffffc(%ebp)
	mov 0xfffffffc(%ebp),%eax
	movl $0x0,(%eax)

(b) Variable access via pointer

int index;
int array[];
 	
void f()
{
  array[index] = 0;
}
	
	mov 0x80495a0,%eax
	movl $0x0,0x80495c0(,%eax,4)

(c) Array access

Figure 2: Unsuitable store instructions.

Of course, not all instructions that write to memory can be actually used to alter a return address. For example, consider the C code fragments and their corresponding machine instructions shown in Figure 2. In the first example (a), the instruction writes to a particular address (`0x8049578`, the address of the variable *global*), which is specified by an immediate operand of the instruction. This store instruction clearly cannot be forced to overwrite an arbitrary memory address. In the other two cases ((b) and (c)), the instruction writes to a location whose address is determined (or influenced) by a value in a register. However, in example (b), the involved register `%eax` has been previously loaded with a constant value (again the address of the variable *global*) that cannot be influenced by the intruder. Finally, even if the attacker can choose the destination address of the store instruction, it is also necessary to be able to control the content that is written to this address. In example (c), the attacker can change the content of the *index* variable before returning control to the application. When the application then performs the array access using the modified *index* variable, which is loaded into register `%eax`, the attacker can write to an (almost) arbitrary location on the stack. However, the constant value 0 is written to this address, making the instruction not suitable to set a return address to the malicious code.

The examples above highlight the fact that even if application code contains many store instructions, only a fraction of them might be suitable to modify return addresses. Even if the original program contains assignments that dereference pointers (or access array elements), it might not be possible to control both the *destination* of the store instruction and its *content*. The possibility of using an assignment operation through a pointer to modify the return address on the stack was previously discussed in [7]. However, the authors did not address the problem that an assignment might not be suitable to perform the actual overwrite. Moreover, if a suitable instruction is found, preparing the environment is often not a trivial task. Consider a situation where an application first performs a number of operations on a set of variables and later stores only the result. In this case, the attacker has to set up the environment so that the result of these operations exactly correspond to the desired value. In addition, one has to consider the effects of modifications to the environment on the control flow of the application.

A simple example is shown in Figure 3. Here, the attacker has to modify the variable *index* to point to the (return) address on the stack that should be overwritten. The value that is written to this location (i.e., the new return address) is determined by calculating the sum of two variables a and b. Also, one has to ensure that $a > 0$ because otherwise the assignment instruction would not be executed.

```
int index, a, b;
int array[];

void f()
{
    if (a > 0)
        array[index] = a + b;
}
```

(a) Possible overwrite

Figure 3: Possibly vulnerable code.

The presented examples serve only as an indication of the challenges that an attacker faces when attempting to manually comprehend and follow different threads of execution through a binary program. To perform a successful attack, it is necessary to take into account the effects of operations on the initial environment and consider different paths of execution (including loops). Also, one has to find suitable store instructions or indirect function calls that can be exploited to recover control. As a result, one might argue that it is too difficult for an attacker to repeatedly make system calls and recover control, which is necessary to perform a successful compromise. In the next section, we show that these difficulties can be overcome.

4 Symbolic Execution

This section describes in detail the static analysis techniques we use to identify and exploit possibilities for regaining control after the invocation of a system call. As mentioned previously, control can be regained when a configuration C is found such that control flow will eventually reach the target address t when program p is executed starting from address s.

Additional constraints are required to make sure that a program execution does not violate the application model created by the intrusion detection system. In particular, system calls may only be invoked in a sequence that is considered legitimate by the intrusion detection system. Also, whenever a system call is invoked, the chain of function return addresses on the stack has to be valid. In our current implementation, we enforce these restrictions simply by requiring that the target address t must be reached from s without making any intermediate system calls. In this way, we ensure that no checks are made by the intrusion detection system before the attacker gets a chance to execute her code. At this point, it is straightforward to have the attack code rearrange the stack to produce a valid configuration (correct chain of function return addresses) and to make system calls in the correct order.

The key approach that we use to find a configuration C for a program p and the two addresses s and t is *symbolic execution* [10]. Symbolic execution is a technique that interpretatively executes a program, using symbolic expressions instead of real values as input. In our case, we are less concerned about the input to the program. Instead, we treat all values that can be modified by the attacker as variables. That is, the execution environment of the program (data, stack, and heap) is treated as the variable input to the code. Beginning from the start address s, a symbolic execution engine interprets the sequence of machine instructions.

To perform symbolic execution of machine instructions (in our case, Intel x86 operations), it is necessary to extend the semantics of these instructions so that operands are not limited to real data objects but can also be symbolic expressions. The normal execution semantics of Intel x86 assembly code describes how data objects are represented, how statements and operations manipulate these data objects, and how control flows through the statements of a program. For symbolic execution, the definitions for the basic operators of the language have to be extended to accept symbolic operands and produce symbolic formulas as output.

4.1 Execution State

We define the execution state S of program p as a snapshot of the content of the processor registers (except the program counter) and all valid memory locations at a particular instruction of p, which is denoted by the program counter. Although it would be possible to treat the program counter like any other register, it is more intuitive to handle the program counter separately and to require that it contains a concrete value (i.e., it points to a certain instruction). The content of all other registers and memory locations can be described by symbolic expressions.

Before symbolic execution starts from address s, the execution state S is initialized by assigning symbolic variables to all processor registers (except the program counter) and memory locations in writable segments. Thus, whenever a processor register or a memory location is read for the first time, without any previous assignment to it, a new symbol is supplied from the list of variables $\{v_l, v_2, v_3, \ldots\}$. Note that this is the only time when symbolic data objects are introduced.

In our current system, we do not support floating point data objects and operations, so all symbols (variables) represent integer values. Symbolic expressions are linear combinations of these symbols (i.e., integer polynomials over the symbols). A symbolic expression can be written as $c_n * v_n + c_{n-1} * v_{n-1} + \ldots + c_1 * v_1 + c_0$ where the

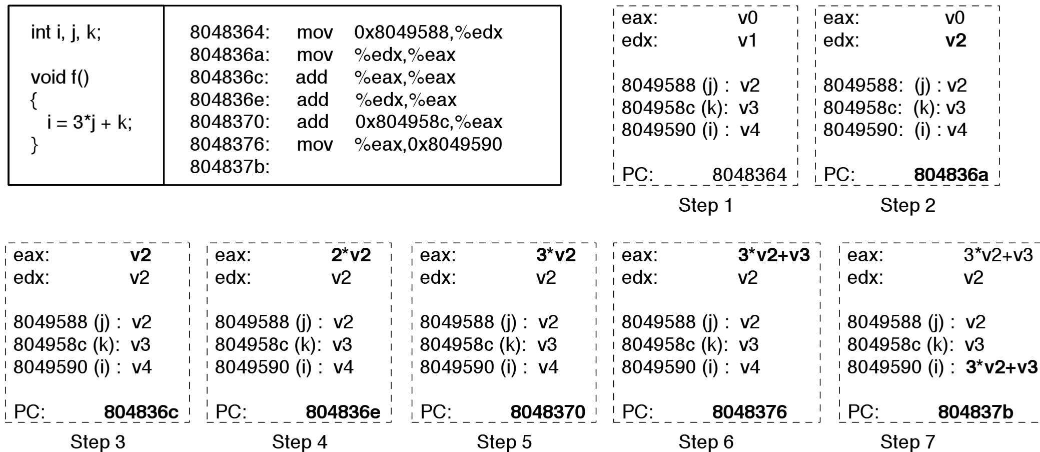

Figure 4: Symbolic execution.

c_i are constants. In addition, there is a special symbol $\perp$ that denotes that no information is known about the content of a register or a memory location. Note that this is very different from a symbolic expression. Although there is no *concrete* value known for a symbolic expression, its value can be evaluated when concrete values are supplied for the initial execution state. For the symbol $\perp$, nothing can be asserted, even when the initial state is completely defined.

By allowing program variables to assume integer polynomials over the symbols v_i, the symbolic execution of assignment statements follows naturally. The expression on the right-hand side of the statement is evaluated, substituting symbolic expressions for source registers or memory locations. The result is another symbolic expression (an integer is the trivial case) that represents the new value of the left-hand side of the assignment statement. Because symbolic expressions are integer polynomials, it is possible to evaluate addition and subtraction of two arbitrary expressions. Also, it is possible to multiply or shift a symbolic expression by a constant value. Other instructions, such as the multiplication of two symbolic variables or a logic operation (e.g., `and`, `or`), result in the assignment of the symbol $\perp$ to the destination. This is because the result of these operations cannot (always) be represented as integer polynomial. The reason for limiting symbolic formulas to linear expressions will become clear in Section 4.3.

Whenever an instruction is executed, the execution state is changed. As mentioned previously, in case of an assignment, the content of the destination operand is replaced by the right-hand side of the statement. In addition, the program counter is advanced. In the case of an instruction that does not change the control flow of a program (i.e., an instruction that is not a jump or a conditional branch), the program counter is simply advanced to the next instruction. Also, an unconditional jump to a certain label (instruction) is performed exactly as in normal execution by transferring control from the current statement to the statement associated with the corresponding label.

Figure 4 shows the symbolic execution of a sequence of instructions. In addition to the x86 machine instructions, a corresponding fragment of C source code is shown. For each step of the symbolic execution, the relevant parts of the execution state are presented. Changes between execution states are shown in bold face. Note that the compiler (`gcc 3.3`) converted the multiplication in the C program into an equivalent series of `add` machine instructions.

4.2 Conditional Branches and Loops

To handle conditional branches, the execution state has to be extended to include a set of constraints, called the *path constraints*. In principle, a path constraint relates a symbolic expression L to a constant. This can be used, for example, to specify that the content of a register has to be equal to 0. More formally, a path constraint is a boolean expression of the form $L \geq 0$ or $L = 0$, in which L is an integer polynomial over the symbols v_i. The set of path constraints forms a linear constraint system.

The symbolic execution of a conditional branch statement starts in a fashion similar to its normal execution, by evaluating the associated Boolean expression. The evaluation is done by replacing the operands with their corresponding symbolic expressions. Then, the inequality (or equality) is transformed and converted into the standard form introduced above. Let the resulting path constraint be called q.

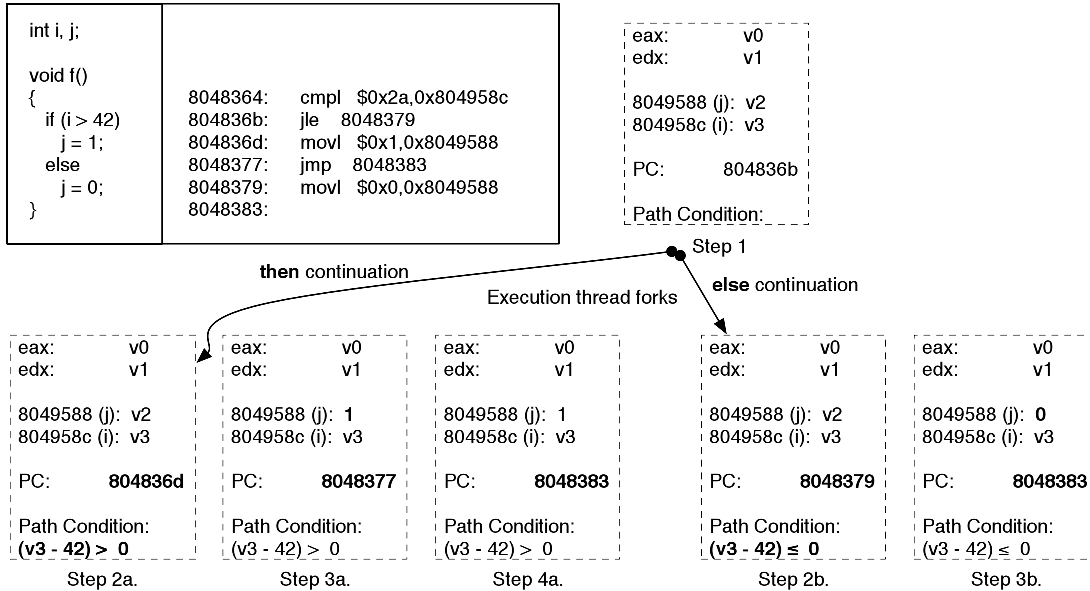

Figure 5: Handling conditional branches during symbolic execution.

To continue symbolic execution, both branches of the control path need to be explored. The symbolic execution forks into two "parallel" execution threads: one thread follows the *then* alternative, the other one follows the *else* alternative. Both execution threads assume the execution state which existed immediately before the conditional statement but proceed independently thereafter. Because the *then* alternative is only chosen if the conditional branch is taken, the corresponding path constraint q must be true. Therefore, we add q to the set of path constraints of this execution thread. The situation is reversed for the *else* alternative. In this case, the branch is not taken and q must be false. Thus, $\neg q$ is added to the path constraints in this execution.

After q (or $\neg q$) is added to a set of path constraints, the corresponding linear constraint system is immediately checked for satisfiability. When the set of path constraints has no solution, this implies that, independent of the choice of values for the initial configuration C, this path of execution can never occur. This allows us to immediately terminate impossible execution threads.

Each fork of execution at a conditional statement contributes a condition over the variables v_i that must hold in this particular execution thread. Thus, the set of path constraints determines which conditions the initial execution state must satisfy in order for an execution to follow the particular associated path. Each symbolic execution begins with an empty set of path constraints. As assumptions about the variables are made (in order to choose between alternative paths through the program as presented by conditional statements), those assumptions are added to the set. An example of a fork into two symbolic execution threads as the result of an `if`-statement and the corresponding path constraints are shown in Figure 5. Note that the `if`-statement was translated into two machine instructions. Thus, special code is required to extract the condition on which a branch statement depends.

Because a symbolic execution thread forks into two threads at each conditional branch statement, loops represent a problem. In particular, we have to make sure that execution threads "make progress" to achieve our objective of eventually reaching the target address t. The problem is addressed by requiring that a thread passes through the same loop at most three times. Before an execution thread enters the same loop for the forth time, its execution is halted. Then, the effect of an arbitrary number of iterations of this loop on the execution state is approximated. This approximation is a standard static analysis technique [2, 13] that aims at determining value ranges for the variables that are modified in the loop body. Since the problem of finding exact ranges and relationships between variables is undecidable in the general case, the approximation naturally involves a certain loss of precision. After the effect of the loop on the execution thread was approximated, the thread can continue with the modified state after the loop.

To approximate the effect of the loop body on an execution state, a *fixpoint* for this loop is constructed. For our purposes, a fixpoint is an execution state F that, when used as the initial state before entering the loop, is equivalent to the final execution state when the loop finishes. In other words, after the operations of the loop body are ap-

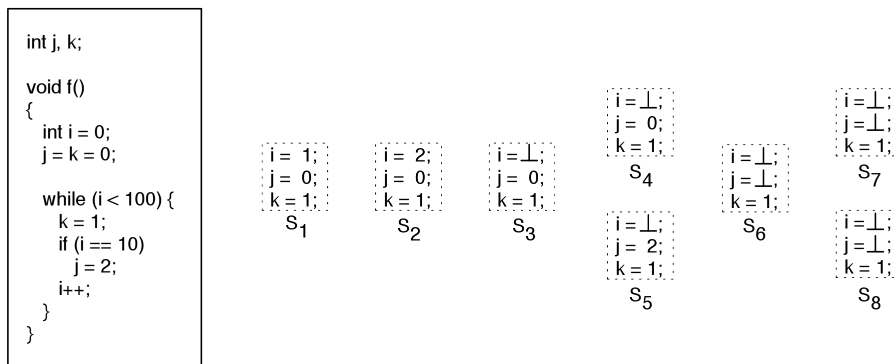

Figure 6: Fixpoint calculation.

plied to the fixpoint state F, the resulting execution state is again F. Clearly, if there are multiple paths through the loop, the resulting execution states at each loop exit must be the same (and identical to F). Thus, whenever the effect of a loop on an execution state must be determined, we transform this state into a fixpoint for this loop. This transformation is often called *widening*. Then, the thread can continue after the loop using the fixpoint as its new execution state.

The fixpoint for a loop is constructed in an iterative fashion as follows: Starting with the execution state S_1 after the first execution of the loop body, we calculate the execution state S_2 after a second iteration. Then, S_1 and S_2 are compared. For each register and each memory location that hold different values (i.e., different symbolic expressions), we assign $\perp$ as the new value. The resulting state is used as the new state and another iteration of the loop is performed. This is repeated until S_i and $S_{(i+1)}$ are identical. In case of multiple paths through the loop, the algorithm is extended by collecting one exit state S_i for each path and then comparing all pairs of states. Whenever a difference between a register value or a memory location is found, this location is set to $\perp$. The iterative algorithm is guaranteed to terminate, because at each step, it is only possible to convert the content of a memory location or a register to $\perp$. Thus, after each iteration, the states are either identical or the content of some locations is made unknown. This process can only be repeated until all values are converted to unknown and no information is left.

An example for a fixpoint calculation (using C code instead of x86 assembler) is presented in Figure 6. In this case, the execution state comprises of the values of the three involved variables i, j, and k. After the first loop iteration, the execution state S_1 is reached. Here, i has been incremented once, k has been assigned the constant 1, and j has not been modified. After a second iteration, S_2 is reached. Because i has changed between S_1 and S_2, its value is set to $\perp$ in S_3. Note that the execution has not modified j, because the value of i was known to be different from 10 at the `if`-statement. Using S_3 as the new execution state, two paths are taken through the loop. In one case (S_4), j is set to 2, in the other case (S_5), the variable j remains 0. The reason for the two different execution paths is the fact that i is no longer known at the `if`-statement and, thus, both paths have to be followed. Comparing S_3 with S_4 and S_5, the difference between the values of variable j leads to the new state S_6 in which j is set to $\perp$. As before, the new state S_6 is used for the next loop iteration. Finally, the resulting states S_7 and S_8 are identical to S_6, indicating that a fixpoint is reached.

In the example above, we quickly reach a fixpoint. In general, by considering all modified values as unknown (setting them to $\perp$), the termination of the fixpoint algorithm is achieved very quickly. However, the approximation might be unnecessarily imprecise. For our current prototype, we use this simple approximation technique [13]. However, we plan to investigate more sophisticated fixpoint algorithms in the future.

To determine loops in the control flow graph, we use the algorithm by Lengauer-Tarjan [12], which is based on dominator trees. Note, however, that the control flow graph does not take into account indirect jumps. Thus, whenever an indirect control flow transfer instruction is encountered during symbolic execution, we first check whether this instruction can be used to reach the target address t. If this is not the case, the execution thread is terminated at this point.

4.3 Generating Configurations

As mentioned in Section 4, the aim of the symbolic execution is to identify code pointers that can be modified to point to the attacker code. To this end, indirect jump and function call instructions, as well as data transfer instructions (i.e., x86`mov`) that could overwrite function re-

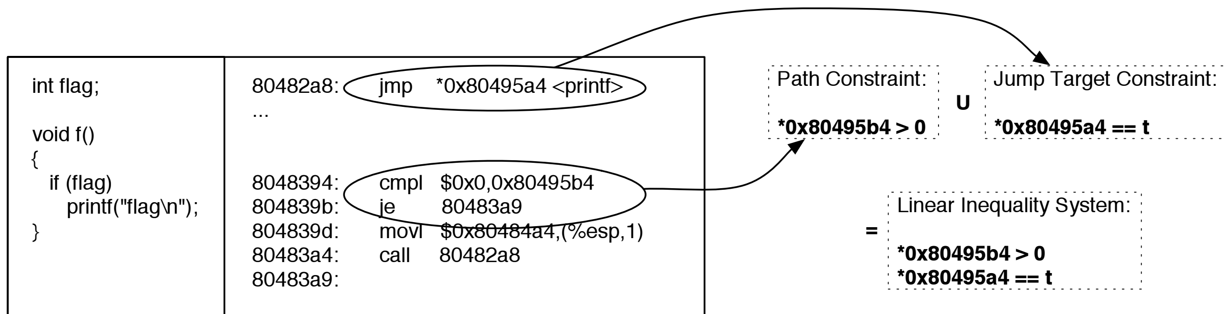

Figure 7: Deriving an appropriate configuration.

turn addresses, are of particular interest. Thus, whenever the symbolic execution engine encounters such an instruction, it is checked whether it can be exploited.

An indirect jump (or call) can be exploited, if it is possible for the attacker to control the jump (or call) target. In this case, it would be easy to overwrite the legitimate target with the address t of the attacker code. To determine whether the target can be overwritten, the current execution state is examined. In particular, the symbolic expression that represents the target of the control transfer instruction is analyzed. The reason is that if it were possible to force this symbolic expression to evaluate to t, then the attacker could achieve her goal.

Let the symbolic expression of the target of the control transfer instruction be called s_t. To check whether it is possible to force the target address of this instruction to t, the constraint $s_t = t$ is generated (this constraint simply expresses the fact that s_t should evaluate to the target address t). Now, we have to determine whether this constraint can be satisfied, given the current path constraints. To this end, the constraint $s_t = t$ is added to the path constraints, and the resulting linear inequality system is solved.

If the linear inequality system has a solution, then the attacker can find a configuration C (i.e., she can prepare the environment) so that the execution of the application code using this configuration leads to an indirect jump (or call) to address t. In fact, the solution to the linear inequality system directly provides the desired configuration. To see this, recall that the execution state is a function of the initial state. As a result, the symbolic expressions are integer polynomials over variables that describe the *initial state* of the system, before execution has started from address s. Thus, a symbolic term expresses the current value of a register or a memory location as a function of the initial values. Therefore, the solution of the linear inequality system denotes which variables of the initial state have to be set, together with their appropriate values, to achieve the desired result. Because the configuration fulfills the path constraints of the current symbolic execution thread, the actual execution will follow the path of this thread. Moreover, the target value of the indirect control transfer instruction will be t. Variables that are not part of the linear inequality system do not have an influence on the choice of the path or on the target address of the control flow instruction, thus, they do not need to be modified.

As an example, consider the sequence of machine instructions (and corresponding C source code) shown in Figure 7. In this example, the set of path constraints at the indirect jump consists of a single constraint that requires `flag` (stored at address 0x80495b4) to be greater than 0. After adding the constraint that requires the jump target (the address of the shared library function `printf` stored at 0x80495a4) to be equal to t, the inequality system is solved. In this case, the solution is trivial: the content of the memory location that holds the jump target is set to t and variable `flag` is set to 1. In fact, any value greater than 0 would be suitable for `flag`, but our constraint solver returns 1 as the first solution.

The handling of data transfer instructions (store operations) is similar to the handling of control transfer instructions. The only difference is that, for a data transfer instruction, it is necessary that the destination address of the operation be set to a function return address **and** that the source of the operation be set to t. If this is the case, the attacker can overwrite a function return address with the address of the attacker code, and, on function return, control is recovered. For each data transfer instruction, two constraints are added to the linear inequation system. One constraint requires that the destination address of the store operation is equal to the function return address. The other constraint requires that the stored value is equal to t. Also, a check is required that makes sure that no system call is invoked between the modification of the function return address and its use in the function epilogue (i.e., on function return). The reason is that the intrusion detection system verifies the integrity of the call stack at each

system call. Note, however, that most applications do not invoke system calls directly but indirectly using library functions, which are usually called indirectly via the PLT. To solve the linear constraint systems, we use the Parma Polyhedral Library (PPL) [1]. In general, solving a linear constraint system is exponential in the number of inequalities. However, PPL uses a number of optimizations to improve the run time in practice and the number of inequalities is usually sufficiently small.

4.4 Memory Aliasing and Unknown Stores

In the previous discussion, two problems were ignored that considerably complicate the analysis for real programs: memory aliasing and store operations to unknown destination addresses.

Memory aliasing refers to the problem that two different symbolic expressions s_1 and s_2 point to the same address. That is, although s_1 and s_2 contain different variables, both expressions evaluate to the same value. In this case, the assignment of a value to an address that is specified by s_1 has unexpected side effects. In particular, such an assignment simultaneously changes the content of the location pointed to by s_2.

Memory aliasing is a typical problem in static analysis, which also affects high-level languages with pointers (such as C). Unfortunately, the problem is exacerbated at machine code level. The reason is that, in a high-level language, only a certain subset of variables can be accessed via pointers. Also, it is often possible to perform alias analysis that further reduces the set of variables that might be subject to aliasing. Thus, one can often guarantee that certain variables are not modified by write operations through pointers. At machine level, the address space is uniformly treated as an array of storage locations. Thus, a write operation could potentially modify any other variable.

In our prototype, we initially take an optimistic approach and assume that different symbolic expressions refer to different memory locations. This approach is motivated by the fact that C compilers (we use `gcc 3.3` for our experiments) address local and global variables so that a distinct expression is used for each access to a different variable. In the case of global variables, the address of the variable is directly encoded in the instruction, making the identification of the variable particularly easy. For each local variable, the access is done by calculating a different offset to the value of the base pointer register (`%ebp`).

Of course, our optimistic assumption might turn out to be incorrect, and we assume the independence of two symbolic expressions when, in fact, they refer to the same memory location. To address this problem, we introduce an additional *a posteriori* check after a potentially exploitable instruction was found. This check operates by *simulating* the program execution with the new configuration that is derived from the solution of the constraint system.

In many cases, having a configuration in which symbolic variables have concrete numerical values allows one to resolve symbolic expressions directly to unambiguous memory locations. Also, it can be determined with certainty which continuation of a conditional branch is taken. In such cases, we can guarantee that control flow will be successfully regained. In other cases, however, not all symbolic expressions can be resolved and there is a (small) probability that aliasing effects interfere with our goal. In our current system, this problem is ignored. The reason is that an attacker can simply run the attack to check whether it is successful or not. If the attack fails, one can manually determine the reason for failure and provide the symbolic execution engine with aliasing information (e.g., adding constraints to specify that two expressions are identical). In the future, we will explore mechanisms to automatically derive constraints such that all symbolic expressions can be resolved to a concrete value.

A store operation to an unknown address is related to the aliasing problem as such an operation could potentially modify any memory location. Again, we follow an optimistic approach and assume that such a store operation does not interfere with any variable that is part of the solution of the linear inequality system (and thus, part of the configuration) and use simulation to check the validity of this assumption.

5 Experimental Results

This section provides experimental results that demonstrate that our symbolic execution technique is capable of generating configurations C in which control is recovered after making a system call (and, in doing so, temporarily transferring control to the application program). For all experiments, the programs were compiled using `gcc 3.3` on a x86 Linux host. Our experiments were carried out on the binary representation of programs, without accessing the source code.

For the first experiment, we attempted to exploit three sample programs that were protected by the intrusion detection systems presented in [4] and [14]. The first vulnerable program is shown in Figure 8. This program starts by reading a password from standard input. If the password is correct (identical to the hard-coded string “secret”), a

command is read from a file and then executed with superuser privileges. Also, the program has a logging facility that can output the command and the identifier of the user that has initially launched the program. The automaton in Figure 9 shows the relevant portion of the graph that determines the sequence of system calls that are permitted by the intrusion detection system. The first `read` system call corresponds to the reading of the password (on line 23), while the `execve` call corresponds to the execution of the command obtained from the file (on line 30). Note the two possible sequences that result because commands can be either logged or not.

```
 1:  #define CMD_FILE "commands.txt"
 2:
 3:  int enable_logging = 0;
 4:
 5:  int check_pw(int uid, char *pass)
 6:  {
 7:     char buf[128];
 8:     strcpy(buf, pass);
 9:     return !strcmp(buf, "secret");
10:  }
11:
12:  int main(int argc, char **argv)
13:  {
14:     FILE *f;
15:     int uid;
16:     char passwd[256], cmd[128];
17:
18:     if ((f = fopen(CMD_FILE, "r")) == NULL) {
19:        perror("error: fopen"); exit(1);
20:     }
21:
22:     uid = getuid();
23:     fgets(passwd, sizeof(passwd), stdin);
24:
25:     if (check_pw(uid, passwd)) {
26:        fgets(cmd, sizeof(cmd), f);
27:        if (enable_logging)
28:           printf("uid [%d]: %s\n", uid, cmd);
29:        setuid(0);
30:        if (execl(cmd, cmd, 0) < 0) {
31:           perror("error: execl"); exit(1);
32:        }
33:     }
34:  }
```

Figure 8: First vulnerable program.

It can be seen that the program suffers from a simple buffer overflow vulnerability in the `check_pw()` function (on line 8). This allows an attacker to inject code and to redirect the control flow to an arbitrary location. One possibility to redirect control would be directly after the check of the password, before the call to the `fgets()` function (on line 26). However, in doing so, the attacker can not modify the command that is being executed because `fgets()` is used to retrieve the command. One solution to this problem could be to first modify the content of the command buffer `cmd`, and then jump directly to the `setuid()` function, bypassing the part that reads the legitimate command from the file. By doing so, however, an alarm is raised by the intrusion detection system that observes an invalid `setuid` system call while expecting a `read`. To perform a classic mimicry attack, the intruder could simply issue a bogus `read` call, but, in our case, such a call would be identified as illegal as well. The reason is that the source of the system call would not be the expected instruction in the application code.

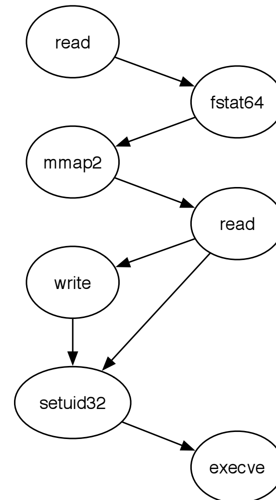

Figure 9: Fragment of automaton that captures permitted system call sequences.

To exploit this program such that an arbitrary command can be executed in spite of the checks performed by the intrusion detection system, it is necessary to regain control *after* the call to `fgets()`. In this case, the attacker could replace the name of the command and then continue execution with the `setuid()` library function. To this end, our symbolic execution engine is used to determine a configuration that allows the attacker to recover control after the `fgets()` call. For the first example, a simple configuration is sufficient in which `enable_logging` is set to 1 and the shared library call to `printf()` is replaced with a jump to the attacker code. With this configuration, the conditional branch (on line 27) is taken, and instead of calling `printf()`, control is passed to the attacker code. This code can then change the `cmd` parameter of the subsequent `execve()` call (on line 30) and continues execution of the original program on line 29. Note that the intrusion detection system is evaded because all system calls are issued by instructions in the application code segment and appear in the correct order.

The buffer overflow in `check_pw()` is used to inject the exploit code that is necessary to set up the environment. After the environment is prepared, control is returned to the original application before `fgets()`, bypassing the

password check routine. Our system is generating actual exploit code that handles the setup of a proper configuration. Thus, this and the following example programs were successfully exploited by regaining control and changing the command that was executed by `execl()`. In all cases, the attacks remained undetected by the used intrusion detection systems [4, 14].

As a second example, consider a modified version of the initial program as shown in Figure 10. In this example, the call to `printf()` is replaced with the invocation of the custom audit function `do_log()`, which records the identifier of the last command issued by each user (with `uid` < 8192). To this end, a unique identifier called `cmd_id` is stored in a table that is indexed by `uid` (on line 6).

```
 1: int enable_logging = 0;
 2: int cmd_id = 0;
 3: int uid_table[8192];
...

 4: void do_log(int uid)
 5: {
 6:     uid_table[uid] = cmd_id++;
 7: }
...

 8: int main(int argc, char **argv)
...
 9:     if (check_pw(uid, passwd)) {
10:         fgets(cmd, sizeof(cmd), f);
11:         if (enable_logging)
12:             do_log(uid);
13:         setuid(0);
14:         if (execl(cmd, cmd, 0) < 0) {
15:             perror("error: execl"); exit(1);
16:         }
17:     }
18: }
```

Figure 10: Second vulnerable program.

For this example, the application was statically linked so that we cannot intercept any shared library calls. As for the previous program, the task is to recover control after the `fgets()` call on line 10. Our symbolic execution engine successfully determined that the assignment to the array on line 6 can be used to overwrite the return address of `do_log()`. To do so, it is necessary to assign a value to the local variable `uid` so that when this value is added to the start address of the array `uid_table`, the resulting address points to the location of the return address of `do_log()`. Note that our system is capable of tracking function calls together with the corresponding parameters. In this example, it is determined that the local variable `uid` is used as a parameter that is later used for the array access. In addition, it is necessary to store the address of the attack code in the variable `cmd_id` and turn on auditing by setting `enable_logging` to a value $\neq 0$.

One might argue that it is not very realistic to store identifiers in a huge table when most entries are 0. Thus, for the third program shown in Figure 11, we have replaced the array with a list. In this example, the `do_log()` function scans a linked list for a record with a matching user identification (on lines 12–14). When an appropriate record already exists, the `cmd_id` field of the `cmd_entry` structure is overwritten with the global command identifier `cmd_id`. When no suitable record can be found, a new one is allocated and inserted at the beginning of the list (on lines 16–21).

```
 1: struct cmd_entry {
 2:     int cmd_id; unsigned int uid;
 3:     struct cmd_entry *next;
 4: };
 5: int enable_logging = 0;
 6: int cmd_id = 0;
 7: struct cmd_entry *cmds = NULL;
...

 8: void do_log(int uid)
 9: {
10:     struct cmd_entry *p;
11:
12:     for (p = cmds; p != NULL; p = p->next)
13:         if (p->uid == uid)
14:             break;
15:
16:     if (p == NULL) {
17:         p = (struct cmd_entry *)
                    calloc(1, sizeof(struct cmd_entry));
18:         p->uid = uid;
19:         p->next = cmds;
20:         cmds = p;
21:     }
22:
23:     p->cmd_id = cmd_id++;
24: }
...

25: int main(int argc, char **argv)
...
26:     if (check_pw(uid, passwd)) {
27:         fgets(cmd, sizeof(cmd), f);
28:         if (enable_logging)
29:             do_log(uid);
30:         setuid(0);
31:         if (execl(cmd, cmd, 0) < 0) {
32:             perror("error: execl"); exit(1);
33:         }
34:     }
35: }
```

Figure 11: Third vulnerable program.

When attempting to find a suitable instruction to direct control flow back to the attacker code, the operation on line 23 seems appropriate. The reason is that this statement assigns the global variable `cmd_id` to the field of a structure that is referenced by the pointer variable `p`. Un-

fortunately, p is not under direct control of the attacker. This is because in the initialization part of the for-loop on line 12, the content of the pointer to the global list head cmds is assigned to p. In addition, the loop traverses the list of command records until a record is found where the uid field is equivalent to the single parameter (uid) of the do_log() function. If, at any point, the next pointer of the record pointed to by p is NULL, the loop terminates. Then, a freshly allocated heap address is assigned to p on line 17. When this occurs, the destination of the assignment statement on line 23 cannot be forced to point to the function return address anymore, which is located on the stack.

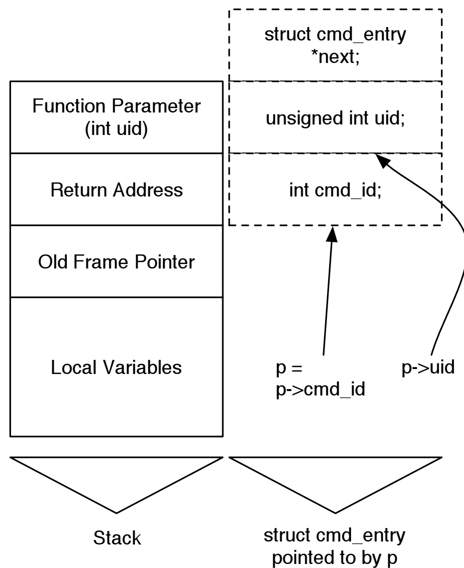

Figure 12: Successful return address overwrite via p.

The discussion above underlines that even if a pointer assignment is found, it is not always clear whether this assignment can be used to overwrite a return address. For this example, our symbolic execution engine discovered a possibility to overwrite the return address of do_log(). This is achieved by preparing a configuration in which the cmds variable points directly to the return address of do_log(). After the content of cmds is assigned to p, p→uid is compared to the uid parameter on line 13. Because of the structure of the cmd_entry record, this comparison always evaluates to *true*. To see why this is the case, refer to Figure 12. The figure shows that when p points to the function's return address, p→uid points to the location that is directly "above" this address in memory. Because of the x86 procedure calling convention, this happens to be the first argument of the do_log() function. In other words, p→uid and the parameter uid refer to the same memory location, therefore, the comparison has to evaluate to *true*. As before, for a successful overwrite, it is necessary to set the value of cmd_id to t and enable auditing by assigning 1 to enable_logging.

Without the automatic process of symbolic execution, such an opportunity to overwrite the return address is probably very difficult to spot. Also, note that no knowledge about the x86 procedure calling convention is encoded in the symbolic execution engine. The possibility to overwrite the return address, as previously discussed, is found directly by (symbolically) executing the machine instructions of the binary. If the compiler had arranged the fields of the cmd_entry structure differently, or if a different calling convention was in use, this exploit would not have been found.

For the second experiment, we used our symbolic execution tool on three well-known applications: apache2, the netkit ftpd server, and imapd from the University of Washington. The purpose of this experiment was to analyze the chances of an attacker to recover control flow in real-world programs. To this end, we randomly selected one hundred addresses for each program that were evenly distributed over the code sections of the analyzed binaries. From each address, we started the symbolic execution processes. The aim was to determine whether it is possible to find a configuration and a sequence of instructions such that control flow can be diverted to an arbitrary address. In the case of a real attack, malicious code could be placed at this address. Note that all applications were dynamically linked (which is the default on modern Unix machines).

Program	Instr.	Success	Failed	
			Return	Exhaust
apache2	51,862	83	12	5
ftpd	9,127	93	7	0
imapd	133,427	88	11	1

Table 1: Symbolic execution results for real-world applications.

Table 1 summarizes the results for this experiment. For each program, the number of code instructions (column "Instr.") are given. In addition, the table lists the number of test cases for which our program successfully found a configuration (column "Success") and the number of test cases for which such a configuration could not be found (column "Failed").

In all successful test cases, only a few memory locations had to be modified to obtain a valid configuration. In fact, in most cases, only a single memory location (a function address in the PLT) was changed. The code that is necessary to perform these modifications is in the order of 100

bytes and can be easily injected remotely by an attacker in most cases.

A closer examination of the failed test cases revealed that a significant fraction of these cases occurred when the symbolic execution thread reached the end of the function where the start address is located (column "Return"). In fact, in several cases, symbolic execution terminated immediately because the randomly chosen start address happened to be a `ret` instruction. Although the symbolic execution engine simulates the run-time stack, and thus can perform function calls and corresponding return operations, a return without a previous function call cannot be handled without additional information. The reason is that whenever a symbolic execution thread makes a function call, the return address is pushed on the stack and can be used later by the corresponding return operation. However, if symbolic execution begins in the middle of a function, when this initial function completes, the return address is unknown and the thread terminates.

When an intruder is launching an actual attack, she usually possesses additional information that can be made available to the analysis process. In particular, possible function return addresses can be extracted from the program's call graph or by examining (debugging) a running instance of the victim process. If this information is provided, the symbolic evaluation process can continue at the given addresses. Therefore, the remaining test cases (column "Exhaust") are of more interest. These test instances failed because the symbolic execution process could not identify a possibility to recover control flow. We set a limit of 1,000 execution steps for each thread. After that, a thread is considered to have exhausted the search space and it is stopped. The reason for this limit is twofold. First, we want to force the analysis to terminate. Second, when the step limit is reached, many memory locations and registers already contain unknown values.

Our results indicate that only a small amount of test cases failed because the analysis engine was not able to identify appropriate configurations. This supports the claim that our proposed evasion techniques can be successfully used against real-world applications.

Program	Steps			Time
	Avg.	Max.	Min.	(in seconds)
apache2	24	131	0	12.4
ftpd	7	62	0	0.3
imapd	46	650	0	1.2

Table 2: Execution steps and time to find configurations.

Table 2 provides more details on the number of steps required to successfully find a configuration. In this table, the average, maximum, and minimum number of steps are given for the successful threads. The results show that, in most cases, a configuration is found quickly, although there are a few outliers (for example, 650 steps for one `imapd` test case). Note that all programs contained at least one case for which the analysis was immediately successful. In these cases, the random start instruction was usually an indirect jump or indirect call that could be easily redirected.

The table also lists the time in seconds that the symbolic execution engine needed to completely check all hundred start addresses (successful and failed cases combined) for each program. The run time for each individual test case varies significantly, depending on the amount of constraints that are generated and the branching factor of the program. When a program contains many branches, the symbolic execution process has to follow many different threads of execution, which can generate an exponential path explosion in the worst case. In general, however, the run time is not a primary concern for this tool and the results demonstrate that the system operates efficiently on real-world input programs.

6 Conclusions

In this paper, we have presented novel techniques to evade two well-known intrusion detection systems [4, 14] that monitor system calls. Our techniques are based on the idea that application control flow can be redirected to malicious code *after* the intruder has passed control to the application to make a system call. Control is regained by modifying the process environment (data, heap, and stack segment) so that the program eventually follows an invalid code pointer (a function return address or an indirect control transfer operation). To this end, we have developed a static analysis tool for x86 binaries, which uses symbolic execution. This tool automatically identifies instructions that can be used to redirect control flow. In addition, the necessary modification to the environment are computed and appropriate code is generated. Using our system, we were able to successfully exploit three sample programs, evading state-of-the-art system call monitors. In addition, we applied our tool to three real-world programs to demonstrate the general applicability of our techniques.

The static analysis mechanisms that we developed for this paper could be used for a broader range of binary analysis problems in the future. One possible application is the identification of configurations for which the current function's return address is overwritten. This might allow

us to build a tool that can identify buffer overflow vulnerabilities in executable code. Another application domain is the search for viruses. Since malicious code is usually not available as source code, binary analysis is a promising approach to deal with this problem. In addition, we hope that our work has brought to attention the intrinsic problem of defense mechanisms that allow attackers a large amount of freedom in their actions.

Acknowledgments

This research was supported by the National Science Foundation under grants CCR-0209065 and CCR-0238492.

References

[1] R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly not closed convex polyhedra and the Parma Polyhedra Library. In *9th International Symposium on Static Analysis*, 2002.

[2] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *4th ACM Symposium on Principles of Programming Languages (POPL)*, 1977.

[3] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, and B. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *IEEE Symposium on Security and Privacy*, 2004.

[4] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *IEEE Symposium on Security and Privacy*, 2003.

[5] S. Forrest. A Sense of Self for UNIX Processes. In *IEEE Symposium on Security and Privacy*, 1996.

[6] D. Gao, M. Reiter, and D. Song. Gray-Box Extraction of Execution Graphs for Anomaly Detection. In *11th ACM Conference on Computer and Communication Security (CCS)*, 2004.

[7] D. Gao, M. Reiter, and D. Song. On Gray-Box Program Tracking for Anomaly Detection. In *13th Usenix Security Symposium*, 2004.

[8] J. Giffin, S. Jha, and B. Miller. Detecting Manipulated Remote Call Streams. In *11th Usenix Security Symposium*, 2002.

[9] J. Giffin, S. Jha, and B.P. Miller. Efficient context-sensitive intrusion detection. In *11th Network and Distributed System Security Symposium (NDSS)*, 2004.

[10] J. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7), 1976.

[11] L. Lam and T. Chiueh. Automatic Extraction of Accurate Application-Specific Sandboxing Policy. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2004.

[12] T. Lengauer and R. Tarjan. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1), 1979.

[13] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Verlag, 1999.

[14] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *IEEE Symposium on Security and Privacy*, 2001.

[15] K. Tan, K. Killourhy, and R. Maxion. Undermining an Anomaly-Based Intrusion Detection System Using Common Exploits. In *5th Symposium on Recent Advances in Intrusion Detection (RAID)*, 2002.

[16] D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *IEEE Symposium on Security and Privacy*, 2001.

[17] D. Wagner and P. Soto. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *9th ACM Conference on Computer and Communications Security (CCS)*, 2002.

[18] C. Warrender, S. Forrest, and B.A. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *IEEE Symposium on Security and Privacy*, 1999.

[19] A. Wespi, M. Dacier, and H. Debar. Intrusion Detection Using Variable-Length Audit Trail Patterns. In *Recent Advances in Intrusion Detrection (RAID)*, 2000.

[20] H. Xu, W. Du, and S. Chapin. Context Sensitive Anomaly Monitoring of Process Control Flow to Detect Mimicry Attacks and Impossible Paths. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2004.