Discovering Application-level Insider Attacks using Symbolic Execution

Karthik Pattabiraman[1], Nithin Nakka[1], Zbigniew Kalbarczyk[1], Ravishankar Iyer[1]

***Abstract**: This paper presents a technique to systematically discover insider attacks in applications. An attack model where the insider is in the same address space as the process and can corrupt arbitrary data is assumed. A formal technique based on symbolic execution and model-checking is developed to comprehensively enumerate all possible insider attacks corresponding to a given attack goal. The main advantage of the technique is that it operates directly on the program code in assembly language and no manual effort is necessary to translate the program into a formal model. We apply the technique to security-critical segments of the OpenSSH application.*

1 Introduction

Insider threats have gained prominence as an emerging and important class of security threats [1, 2]. An insider is a person who is part of the organization and either steals secrets or subverts the working of the organization by exploiting hidden system flaws for malicious purposes. For example, a web browser may have a malicious plugin that overwrites the address bar with the address of a phishing website. Or a disgruntled programmer may plant a logical flaw in a banking application that allows an external user to fraudulently withdraw money. Both are examples of how a trusted insider can compromise an application and subvert it for malicious purposes.

This paper considers *application-level* insider attacks. We define an application-level insider attack as one in which a malicious insider attempts to overwrite one or more data items in the application, in order to achieve a specific attack goal. The overwriting may be carried out by exploiting existing vulnerabilities in the application (e.g. buffer overflows), by introducing logical flaws in the application code or through malicious third-party libraries. It is also possible (though not required) to launch insider attacks from a malicious operating system or higher-privileged process. Application-level insider attacks are particularly insidious because, (1) by attacking the application an insider can evade detection by mimicking its normal behavior (from the point of view of the system), and (2) to attack the application, it is enough for the insider to have the same privilege as that of the application (assuming a flat address space where all modules have equal privileges), whereas attacking the network or operating system may require super-user privileges.

[1]Center for Reliable and High-Performance Computing (CRHC), University of Illinois at Urbana-Champaign (UIUC), Urbana, IL. *email: {pattabir, nakka, kalbarcz, rkiyer} @uiuc.edu*

Before defending against insider attacks, we need a model for reasoning about insiders. Previous work has modeled insider attacks at the network and operating system (OS) levels using higher-level formalisms such as attack graphs [3] and process calculi [4]. However, modeling application-level insider attacks requires analysis of the application's code as an insider has access to the application and can hence launch attacks on the application's implementation. Higher-level models are too coarse grained to enable reasoning about attacks that can be launched at the application code level. Further, higher-level models typically require application vulnerabilities to be identified upfront in order to reason about insider attacks.

This paper introduces a technique to formally model application-level insider attacks on the application code (expressed in assembly language). The advantage of modeling at the assembly level is that the assembly code includes the program, libraries, and any state added by the compiler (e.g. stack pointer and return addresses), and enables accurate reasoning about all *software-based* insider attacks.

The proposed technique uses a combination of symbolic execution and model checking to systematically enumerate *all* possible insider attacks in a given application corresponding to an attack goal. The technique can be automatically deployed on the application's code and no formal specifications need to be provided other than generic specifications about the attacker's end goal(s).

The value of the analysis performed by the proposed technique is that it can expose non-intuitive cases of insider attacks that may be missed by manual code inspection. This is because the technique exhaustively considers corruptions of data items used in the application (under a given input), and enumerates all corruptions that lead to a successful attack (based on the specified attack goal). The results of the analysis can be used to guide the development of defense mechanisms (e.g. assertions).

We have implemented the proposed technique as a tool, *SymPLAID*, which directly analyzes MIPS-based assembly code. The tool identifies for each attack, (1) The program point at which the attack must be launched, (2) The data item that must be overwritten by the attacker, and (3) The value that must be used for overwriting the data item in order to carry out the attack.

SymPLAID builds on our earlier tool, SymPLFIED [5], used to evaluate the effect of transient errors on the application. SymPLFIED groups individual errors into a single abstract class (*err)*, and considers the effect of the entire class of errors on the program. This is because in the case of randomly occurring errors, we are more interested in the propagation of the error rather than the precise set of circumstances that caused the error. In contrast, security attacks are launched by an intelligent adversary and hence it is important to know precisely what values are corrupted by the attacker in order to design efficient defense mechanisms against the attack(s). Therefore, SymPLAID was built from the ground up to emphasize precision in terms of identifying the specific conditions for an attack. Thus, rather than abstracting the attacker's behavior into a single class, the effect of each value corruption is considered individually, and its propagation is tracked in the program. The key contributions of the paper are:

1. Introduces a formal model for reasoning about application-level insider attacks at the assembly-code level,
2. Shows how application-level insiders may be able to subvert the execution of the application for malicious purposes,
3. Describes a technique to automatically discover *all* possible insider attacks in an application using symbolic execution and model checking,
4. Demonstrates the proposed techniques using a case-study drawn from the authentication module of the OpenSSH application [6].

2 Insider Attack Model

This section describes the attack model for insider attacks and an example scenario for an insider attack.

2.1 Characterization of Insider

Capabilities: The insider is a part of the application and has unfettered access to the program's address space. This includes the ability to both read and write the program's memory and registers. However, we assume that the insider cannot modify the program's code, which is reasonable since in most programs the code segment is marked read-only after the program is loaded. An attacker may get into the application (and become an insider) in one or more of the following ways:

1. By a logical loophole in the application planted by a disgruntled or malicious programmer,
2. Through a malicious (or buggy) third-party library loaded into the address space of the application,
3. By exploiting known security loopholes such as buffer overflow attacks and planting the attack code,
4. By overwriting the process's registers or memory from another process (with higher privilege) or debugger,
5. Through a security vulnerability in the operating system or virtual machine (if present)

In each of the above scenarios, the insider can corrupt the values of either memory locations or registers while the application is executing. The first three scenarios only require the insider to have the same privileges as the applications, while the last two require higher privileges.

Goal: The attacker's goal is to subvert the application to perform malicious functions on behalf of the attacker. However, the attacker wants to elude detection or culpability (as far as possible), so the attacker's code may not directly carry out the attack, but may instead overwrite elements of the program's data or control in order to achieve the attacker's aims. From an external perspective, it will appear as though the attack originated due to an application malfunction, and hence the attack code will not be blamed. Therefore, the attacker can execute code to overwrite crucial elements of the program's data or control elements.

It is assumed that the attacker does not want to crash the application, but wants to subvert its execution for some malicious purpose. The attack is typically launched only under a specific set of inputs to the program (known to the attacker), and the input sequence that launches the attack is indistinguishable from a legitimate input for the program. Even if the insider is unable to launch the attack by himself/herself, he/she may have a colluding user who supplies the required inputs to launch the attack.

2.2 Attack Scenario

Figure 1 shows an example attack scenario where the insider has planted a "logic bomb" in the application which is triggered under a specific set of inputs. Normal users are unlikely to accidentally supply the trigger sequence and will be able to use the application without any problems. However, a colluding user knows about the time-bomb and supplies the trigger sequence as input. Perimeter based protection techniques such as firewalls will not notice anything unusual as the trigger sequence is in-

distinguishable from a regular input for all practical purposes. However, the input will trigger the time-bomb in the application thereby launching the security attack on behalf of the insider.

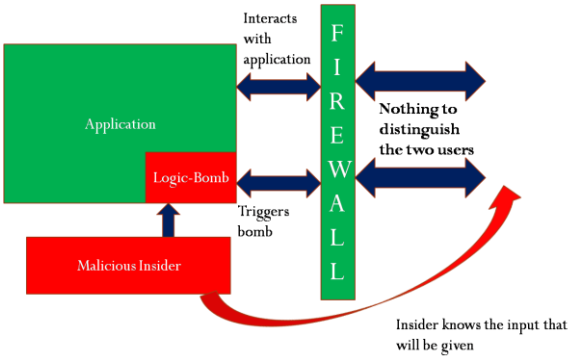

Figure 1: Attack scenario of an insider attack

2.3 Problem Definition

The problem of attack generation from the insider's point of view may be summed up as follows: "If the input sequence to trigger the attack is known (AND) the attacker's code is executed at specific points in the program, what data items in the program should be corrupted and in what way to achieve the attack goal?"

This paper develops a technique to automatically discover conditions for insider attacks in an application given (i) the inputs to trigger the attack (e.g. a specific username as input), (ii) the attacker's objective stated in terms of the final state of the application (e.g. to allow a particular user to log in with the wrong password) and (iii) the attacker's capabilities in terms of the points from which the attack can be launched (e.g. within a specific function). The analysis identifies both the target data to be corrupted and what value it should be replaced with to achieve the attacker's goal. To facilitate the analysis, the following assumptions are made about the attacker by the technique. (1) Only one value can be corrupted, but the corrupted value can be any valid value. This assumption ensures that the footprint of the attack is kept small which makes it easier to evade detection (from a defense technique) and (2) Corruption is only allowed at fixed program points. This assumption reflects the fact that an insider may be able launch their attacks only at fixed program points – for example, where the untrusted library function is called.

3 Example Code and Attacks

This section considers an example code fragment to illustrate the attack scenario in Section 2.2. The example is motivated by the OpenSSH program [6], but is not the actual code extract (we consider the real OpenSSH application in Section 5).

Figure 2 shows an example code fragment containing the *authenticate* function. The *authenticate* function reads the values of the system password and the user password into the *tmp* buffer. It copies the value of the system password into the *src* buffer and the value of the user password into the *dst* buffer. It then compares the values in the *src* and *dst* buffers and if they match, it returns the value 1 (authenticated). Otherwise it returns the value 0 (unauthenticated) to the caller.

```
int authenticate(void* src, void* dst, void* temp, int len){
    1: readSystemPassword(temp);
    2: strncpy(src, temp, len)
    3: readUserPassword(temp);
    4: strncpy(dst, temp, len);
    5: if (! strncmp(dst, src, len) ) return 1;
    return 0;
}
```

Figure 2: Code of the authenticate function

We take the attacker's perspective in coming up with insider attacks on the code in Figure 2. The attacker's goal is to allow a colluding user (who may be the same person as the attacker) to be validated even if he/she has entered the wrong password. The following assumptions are made in this example, for simplicity of explanation:

1. The attack can be invoked only within the body of the *authenticate* function.
2. The attacker can overwrite the value of registers and local variables, but not global variables and heap buffers (due to practical limitations such as not knowing the exact address of global variables and dynamic memory objects).
3. The attack points are immediately before the function calls within the *authenticate* function, i.e., the arguments to any of the functions called by the *authenticate* function may be overwritten prior to the function call.

Table 1 shows the set of all possible attacks the attacker could launch in the above function. A particularly interesting attack found is presented in row 6 of Table 1, where the *dst* argument of the *strncpy* function was set to overlay the *src* string in memory. This replaces the first character of the *src* string with '\0', effectively converting it to a NULL string. The *dst* string also becomes NULL as the *dst* buffer is not filled by the *strncpy* function (we assume that it has initially been filled with all zeroes). The two strings will match when compared and the *authenticate* function will return '1'. This allows the colluding user to be authenticated.

Table 1: Insider attacks on the authenticate function

Program Point	**Variable to be corrupted**	**Corrupted value of variable**	**Comments/Explanation**
strncmp point (line 5)	dst	src buf	The *src* buffer is compared with itself
	src	dst buf	The *dst* buffer is compared with itself
	src	temp buf	The *dst* buffer is compared with the *temp* buffer which contains the same string
	len	<= 0	The strncmp function terminates early and returns 0 (the strings are identical)
strncpy point (line 4)	temp	src buf	This copies the string in the *src* buffer to the *dst* buffer, thereby ensuring that the strings match
	dst	srcBuf – strlen(buf)	This writes a '\0' character in the *src* buffer, effectively converting it to a empty string. The *dst* buffer is also empty as it is not initialized (assuming it is initially set to all zeroes), and hence the strings match.
readUser Pass-word point (line 3)	temp	dst buf	The temp buffer originally contains the system password. Due to the attack, the value in the *temp* buffer is not replaced with the user password. Therefore, the system password is copied to the *dst* buffer, which matches the contents of the *src* buffer i.e., the system password.
	temp	Any unused location in memory	

As Table 1 shows, discovering all possible insider attacks manually (by inspection) is cumbersome and non-trivial even for the modestly sized piece of code that is considered in Figure 2. Therefore, we have developed a tool, SymPLAID, to automatically generate insider attacks scenarios. The attacks in Table 1 were discovered by SymPLAID. Although the tool works on assembly language programs, we have shown the program as C-language code in Figure 2 for simplicity. We have validated the attacks shown in Table 1 using the GNU debugger (*gdb*) to corrupt the values of chosen variables in the application on an AMD machine running the Linux operating system. All the attacks in Table 1 were found to be successful i.e. they led to the user being authenticated in spite of providing the wrong password.

The attacks in Table 1 contain both "obvious attacks" as well as surprising corner cases. It can be argued that finding obvious attacks is not very useful as they are likely to be revealed by manual inspection of the code. However, the power of the proposed technique is that it can reveal *all* such attacks on the code, whereas a human operator may miss one or more attacks. This is especially important from the developer's perspective, as *all* the security holes in the application need to be plugged before it can be claimed that the application is secure (as all the attacker needs to exploit is a single vulnerability). Moreover, the ability to discover corner-case attacks is the real benefit of using an automated approach.

The attacks discovered by SymPLAID can be used to guide the development of defense mechanisms. For example, for the attacks discovered in Table 1, we insert runtime checks at the following points:

1. Before the call to the *strncmp* function to ensure that the *src* and *dst* buffers of the *strncmp* function do not overlap with each other or with the *temp* buffer in terms of physical locations. This prevents attacks in rows 1 to 4 of Table 1.
2. After the call to the *readUserPassword* function in line 3 to ensure that the *temp* buffer is non-empty. This prevents attacks in the rows 7 and 8 of Table 1.
3. Before the call to the *strncpy* function to ensure that neither the *temp* buffer nor the *dst* buffer overlap with the *src* buffer. This prevents attacks in the rows 5 and 6 of Table 1.

Figure 3 shows the code in Figure 2 with the checks inserted as *assert* statements. It is assumed that the checks are themselves immune to attack from an insider.

```
int authenticate(void* src, void* dst, void* temp, int len){
    1: readSystemPassword(temp);
    2: strncpy(src, temp, len)
    3: readuserPassword(temp);
 assert( isNotEmpty(temp) );  assert( noOverlap(temp, src) and noOverlap(temp, dst) )
    4: strncpy(dst, temp, len);
 assert( noOverlap(src, dst) and noOverlap(src, temp) ); assert( len > 0 );
    5: if (! strncmp(dst, src, len) ) return 1;
    return 0;
}
```

Figure 3: Code of authenticate function with assertions

4 Technique and Tool

This section describes the key techniques used in the automation and the design of a tool to automatically discover insider attacks in an application.

4.1 Symbolic Execution Technique

We represent an insider attack as a corruption of data values at specific points in the program's execution i.e. attack points. The attack points are chosen by the program developer based on knowledge of where an insider can attack the application. For example, all the places where the application calls an untrusted third-party library are attack points as an insider can launch an attack from these points. In the worst-case, every instruction in the application can be an attack point.

The program is executed with a known (concrete) input, and when one of the specified execution points is reached, a single variable[2] is chosen from the set of all variables in the program and assigned a symbolic value (i.e. not a concrete value). The program's execution is continued with the symbolic value for the chosen variable. All other variables in the program are unchanged. The above procedure is repeated exhaustively for each data value in the program at each of the specified attack points. This allows enumeration of all insider attacks on a given program.

The key technique used to comprehensively enumerate insider attacks is *symbolic execution-based model checking*. This means that the program is executed with a combination of concrete values and symbolic values, and model-checking is used to "fill-in" the symbolic values as and when needed. Symbolic values are treated similar to concrete values in arithmetic and logical computations performed in the system. The main difference is in how branches and memory accesses based on expressions involving symbolic values are handled. When a memory access is performed with a symbolic expression as the address operand, the execution of the program is forked and the symbolic expression is equated to a different memory address in each fork. The value stored at the address is read or written in the corresponding fork and the program's execution is continued. Once the symbolic value has been assigned to an address, all expressions involving the symbolic value in the state are concretized.

Similarly, in the case of branches involving symbolic expressions, the program execution is forked at the branch point. The branch condition is added as a constraint to the first fork, while the negation of the condition is added as a constraint to the second fork. For each program fork encountered above, the model checker checks whether (1) The fork is a viable one, based on the past constraints of the symbolic expressions, and (2) whether the fork leads to a desired outcome (of the attacker). If these two conditions are satisfied, the model checker will print the state of the program corresponding to the fork i.e. attack state.

As in most model-checking approaches, the number of states explored can be exponential in the size of the program and its address space. However, very few of the states explored by the model-checker will satisfy the attacker's goal(s). Hence, the model-checker can prune branches of the search tree once it is clear that the branch will not lead to a state satisfying the goal. This is the key to the scalability of the approach, and underlies the importance of specifying an attack goal for the insider.

4.2 SymPLAID Tool

The symbolic execution technique described in the previous section has been implemented in an automated tool – SymPLAID (*Symbolic Program Level Attack Injection and Detection*). This is based on our earlier tool, SymPLFIED, used to study the effect of transient errors on programs [5].

---

[2] We use the generic term variable to refer to both registers and memory locations in the program.

SymPLAID accepts the following inputs: (1) an assembly language program along with libraries (if any), (2) a set of pre-defined inputs for the program, (3) a specification of the desired goal of the attacker (expressed as a formula in first-order logic) and (4) a set of attack points in the application. It generates a comprehensive set of insider attacks that lead to the goal state. For each attack, SymPLAID generates both the location (memory or register) to be corrupted as well as the value that must be written to the location by the attacker.

SymPLAID directly parses and interprets assembly language programs written for a MIPS processor. The current implementation supports the entire range of MIPS instructions, including (1) arithmetic/logical instructions, (2) memory accesses (both aligned and unaligned) and (3) branches (both direct and indirect). However, it does not support system calls. The lack of system call support is compensated for by the provision of native support for input/output operations. Floating point operations are also not considered by SymPLAID. This is not a bottleneck as floating-point operations are typically not used by security-critical code in applications.

SymPLAID is implemented using Maude, a high-performance language and system that supports specification and programming in rewriting logic [7]. SymPLAID models the execution semantics of an assembly language program using both equations and rewriting rules. Equations are used to model the concrete semantics of the machine, while rewriting rules are used for introducing non-determinism due to symbolic evaluation. SymPLAID maintains precise dependencies both in terms of arithmetic and logical constraints and solves the constraints without incurring false-positives. This is the biggest difference between SymPLAID and SymPLFIED [5], which aggregates symbolic values into a single class and hence incurs false-positives.

5 Case Study: OpenSSH Authentication Module

To evaluate the SymPLAID tool on a real application, we considered a reduced version of the OpenSSH application [6] involving only the user-authentication part. This is because SymPLAID does not support all the features used in the complete SSH application, e.g. system calls. We retain the core functions in the authentication part of OpenSSH with little or no modifications, and replace the more complex ones with stub versions – i.e. simplified functions that approximate the behavior of their original versions. We also replace the system calls with stubs. The reduced version, the *authentication module*, consists of about 250 lines of C code and emulates the behavior of the SSH application starting from the point after the user enters his/her username and password to the point that he/she is authenticated or denied authentication by the system (we consider only password-based authentication).

We ran SymPLAID on the authentication module after compiling it to MIPS assembly using the *gcc* compiler. As before, the goal is to find insider attacks that will allow the user to be authenticated. It is assumed that the insider can overwrite the value of any register prior to executing any instruction within the authentication module. The input to the authentication module is the username and password. The username may or may not be a valid username in the system, and the password may or may not be correct. These lead to four possible categories of inputs.

SymPLAID discovered attacks corresponding to the categories where an invalid username is supplied with a valid password (for the application) and where a valid username is supplied with an incorrect password. An example of an attack where the invalid username is supplied is considered. Due to space constraints, the other attacks are not described and may be found in the technical report [8].

5.1 Example Attack: Invalid User-name

The authentication part of SSH works as follows: when the user enters his/her name, the program first checks the user-name against a list of users who are allowed to log into the system. If the user is allowed to log into the system, the user record is assigned to a data-structure called an *authctxt* and the user details are stored into the *authctxt* structure. If the name is not found on the list, the record is assigned to a special data-structure in memory called as *fake*. *fake* is also an *authctxt* structure, except that it holds a dummy username and password. This ensures that there is no observable difference in the time it takes to process legitimate and illegitimate users (which may enable attackers to learn if a username is valid by repeated attempts).
In order to prevent potential attackers from logging on by providing this dummy password, the *authctxt* structure has an additional field called *valid*. This field is set to *true* only for legitimate *authctxt* records i.e. those for which the username is in the list of valid users for the system. The *fake* structure has the *valid* field set to *false* by default. In order for the authentication to succeed, the encrypted value of the user password must match the (encrypted) system password, *and* the valid flag of the *authctxt* record must be set to 1. Figure 4 shows the *auth_password* function that performs the above checks. The function first calls the *sys_auth_passwd* to check if the passwords match, and then checks if the *valid* flag is set in the *authctxt* record. Only if both conditions are true will the function return 1 (authenticated) to its caller.

```
int sys_auth_passwd(Authctxt *authctxt, const char *password)  {
1:          struct passwd *pw = authctxt->pw;
2:          char *encrypted_password;
3:          char *pw_password = authctxt->valid ?
4:                      shadow_pw(pw) : pw->pw_passwd;
5:          if (strcmp(pw_password, "") == 0 &&
6:              strcmp(password, "") == 0)
7:                  return (1);
8:          encrypted_password = xcrypt(password,
9:                      (pw_password[0] && pw_password[1]) ?
10:                      pw_password : "xx");
11:   return (strcmp(encrypted_password, pw_password) == 0);
}

int auth_password(Authctxt *authctxt, const char *password) {
12:             int permit_empty_passwd = 0;
13:         struct passwd * pw = authctxt->pw;
14:         int result, ok = authctxt->valid;
15:         if (*password == '\0' && permit_empty_passwd == 0)
16:                 return 0;
17:         result = sys_auth_passwd(authctxt, password);
18:         if (authctxt->force_pwchange)
19:                 disable_forwarding();
20:         return (result && ok);
}
```

Figure 4: SSH code fragment corresponding to the attack

An insider can launch an attack by setting the *valid* flag to true for the *fake* authctxt structure. This will authenticate a user who enters an invalid user name, but enters the password stored in the *fake* structure. The password in the *fake* structure is a string that is hardcoded into the program. To mimic this attack, we supply an invalid user-name

and a password that matches the *fake* (dummy) password. We expected SymPLAID to find the attack where the insider overwrites the valid flag of the *fake* structure. SymPLAID found this attack, but it also found other interesting attacks.

We consider an example of an attack found by SymPLAID. The attack occurs in the *sys_auth_password* function, at line 11 before the call to the *strcmp* function (in Figure 4). At this point, the insider corrupts the value of the stack pointer (stored in register $30 in the MIPS architecture) to point within the stack frame of the caller function, namely *auth_password*. When the *strcmp* function is called, it pushes the current frame pointer onto the stack, increments the stack pointer and sets its frame pointer to be equal to the value of the stack pointer (corrupted by the attacker).

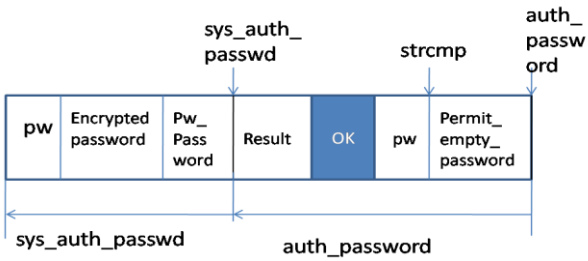

Figure 5: Stack layout when strcmp() is called

Figure 5 shows the stack layout when the function is called (only the variables relevant to the attack are shown). The top-row of Figure 5 shows the frame-pointers of the functions on the stack due to the attack. Observe that the attack causes the stack frame of the *strcmp* function to overlap with that of the *auth_password* function. The *strcmp* function is invoked with the addresses of the *encrypted_pasword* and the *pw_password* buffers in registers *$3* and *$4* (function arguments are passed in registers on MIPS processors). The function copies the contents of these registers to locations within its stack frame at offsets of 4 and 8 respectively from its frame pointer. This overwrites the value of the local variable *ok* in the *auth_password* function with a non-zero value (since both buffers are at non-zero addresses). When the *strcmp* function returns, the value of $30 is restored to the frame pointer of *sys_auth_passwd*, which in turn returns to the *auth_password* function. The *auth_password* function checks if the result returned from *sys_auth_password* is non-zero and if the *ok* flag is non-zero. Both conditions are satisfied, so it returns the value 1 to its caller, and the user is authenticated successfully by the system.

5.2 Performance Results

The model-checking task is highly parallelizable and can be broken into independent sub-tasks, with each sub-task considering attacks in a different code region of the application. The authentication module consists of about 500 assembly language instructions, and the task was broken up into 50 parallel sub-tasks each of which analyzes 10 instructions in the program. We executed the sub-tasks on a parallel cluster consisting of dual-processor AMD Opteron nodes, each of which has 2 GB RAM. The maximum time allowed for each task was capped at 48 hours (2 days).

The total time taken to execute all sub-tasks is at most 36 days. However, the task finished in less than 2 days due to the highly parallel nature of the search task. While

the running time seems high, it is not a concern as the goal is to discover all potential insider attacks (in a reasonable time) and to find protection mechanisms against them.

6 Related Work

Insider attacks have traditionally been modeled at the network level. Philips and Swiler [9] introduced the attack graph model to represent the set of all possible attacks that can be launched in a network. Ammann et al. [10] introduce a model-checking based technique to automatically find attacks starting from a known goal state of the attacker. Sheyner et al. generalize this technique to generate all possible attack paths, thereby generating the entire attack graph [11]. Chinchani et al. [3] present a variant of attack graphs called key-challenge graphs to represent insider attacks, and use model-checking to generate all possible insider attacks in a network.

Insider attacks have been modeled at the operating system level by Probst et al. [4]. In this model, applications are represented as sets of processes that can access sets of resources in the system. An insider is modeled as a malicious process in the system.

Attack-graphs and process graphs are too coarse grained for representing application-level attacks, and hence we directly analyze the application's code. Further, we do not require the developer to provide a formal description of the system being analyzed, which can require significant effort. Since we analyze the application's code directly, we can model attacks both in the design and implementation of the application. This is important as an insider typically has access to the application's source code, and can launch low-level attacks on its implementation.

Symbolic execution is a well-explored technique to find program errors [12]. Recently, it has also been used to find security vulnerabilities in applications [13-16]. Symbolic techniques are typically concerned with generating application inputs to exploit known or unknown vulnerabilities. In contrast, our technique attempts to generate attacks under a given input, assuming that the attacker is already present in the application. Further, the attacks found using our technique do not require the application to have an exploitable vulnerability (e.g. buffer overflows), but can be launched by a malicious insider in the system.

Fault-injection is an experimental technique to assess the vulnerability of computer systems to random events or faults [17]. Fault-injection has also been used to expose security vulnerabilities in applications. Fault-injection studies [18, 19] into commonly used cryptographic systems have shown that transient faults can weaken the guarantees provided by these systems. The main difference between these studies and ours is that our technique can be applied for any general security-critical system, and not just crypto-systems. Xu et al. [20] consider the effect of transient errors (single-bit flips) in instructions on application security. Govindavajhala and Appel [21] explore the effects of transient errors on the security of the Java virtual machine, assuming the attacker can execute a specially crafted application. The main difference between these techniques and our technique is that we consider all possible attacks on the application, and are not restricted to injecting single bit-flips. Further, we do not require the attacker to execute specially crafted programs as assumed by [21].

7 Conclusions

This paper presented a novel approach to discover insider attacks in applications. An automated technique to find all possible insider attacks on application code is presented. The technique uses a combination of symbolic execution and model-checking to systematically enumerate insider attacks for a given goal of the attacker. We have

implemented the technique in the SymPLAID tool, and demonstrate it using the code segments corresponding to the authentication part of the OpenSSH program.

Acknowledgments: This work was supported in part by NSF grants CNS-0406351 (Next-generation Software), CNS-05-24695, CNS-05-51665, the Gigascale Systems Research Center (GSRC/MARCO), and Boeing Corporation as part of Boeing Trusted Software Center at the Information Trust Institute.

References

1. Randazzo, M.R., et al., *Insider Threat Study: Illicit Cyber Activity in the Banking and Finance Sector*. 2004, ERT Coordination Center/Software Engineering Institute: Philadelphia, PA. p. 25.
2. Keeney, M.M. and E.F. Kowalski, *Insider Threat Study: Computer System Sabotage in Critical Infrastructure Sectors*. 2005, CERT/CC: Philadelphia, PA.
3. Chinchani, R., et al., *Towards a Theory of Insider Threat Assessment*, in *Proceedings of the 2005 International Conference on Dependable Systems and Networks*. 2005, IEEE Computer Society.
4. Probst, C.W., R.R. Hansen, and F. Nielson, *Where can an Insider Attack ?*, in *Formal Aspects in Security and Trust*. 2007, Springer Berlin / Heidelberg. p. 127-142.
5. Pattabiraman, K., N. Nakka, and Z. Kalbarczyk. *SymPLFIED: Symbolic Program Level Fault-Injection and Error-Detection Framework*. in *International Conference on Dependable Systems and Networks (DSN)*. 2008.
6. OpenSSH Development Team., *OpenSSH 4.21*. 2004.
7. Clavel, M., et al. *The Maude 2.0 System*. in *Rewriting Technologies and Applications*. 2001: Springer.
8. Pattabiraman, K., et al., *Discovering Application-level Insider Attacks using Symbolic Execution*, in *CRHC Technical Report*. 2008, UIUC: Champaign, IL.
9. Phillips, C. and L.P. Swiler, *A graph-based system for network-vulnerability analysis*, in *Proceedings of the 1998 workshop on New security paradigms*. 1998, ACM: Charlottesville, Virginia,.
10. Ammann, P., D. Wijesekera, and S. Kaushik, *Scalable, graph-based network vulnerability analysis*, in *Proceedings of the 9th ACM conference on Computer and communications security*. 2002, ACM: Washington, DC, USA.
11. Sheyner, O., et al., *Automated Generation and Analysis of Attack Graphs*, in *Proceedings of the 2002 IEEE Symposium on Security and Privacy*. 2002, IEEE Computer Society.
12. King, J.C., *Symbolic execution and program testing.* Commun. ACM, 1976. **19**(7): p. 385-394.
13. Costa, M., et al., *Bouncer: securing software by blocking bad input*, in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. 2007, ACM: Stevenson, Washington,.
14. Kruegel, C., et al., *Automating mimicry attacks using static binary analysis*, in *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*. 2005, USENIX,: Baltimore, MD.
15. Molnar, D.A. and D. Wagner, *Catchconv: Symbolic execution and run-time type inference for integer conversion errors*. 2007, EECS Department, University of California, Berkeley.
16. Cadar, C., et al., *EXE: automatically generating inputs of death*, in *Proceedings of the 13th ACM conference on Computer and communications security*. 2006, ACM: Virginia.
17. Hsueh, M.-C., T.K. Tsai, and R.K. Iyer, *Fault Injection Techniques and Tools.* IEEE Computer, 1997. **30**(4): p. 75-82.
18. Boneh, D., R. DeMillo, and R.J. Lipton. *On the importance of checking crypto-graphic protocols for faults.* in *Advances in Cryptgraphy (EuroCrypt)*. 1997: Springer.
19. Kocher, P.C., J. Jaffe, and B. Jun, *Differential Power Analysis*, in *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*. 1999, Springer..
20. Xu, J., et al., *An Experimental Study of Security Vulnerabilities Caused by Errors*, in *Proceedings of International Conference on Dependable Systems and Networks (DSN)*, 2001.
21. Govindavajhala, S. and A.W. Appel, *Using Memory Errors to Attack a Virtual Machine*, in *Proceedings of the 2003 IEEE Symposium on Security and Privacy*. 2003, IEEE.