Static Detection of Vulnerabilities in x86 Executables

Marco Cova   Viktoria Felmetsger   Greg Banks
Giovanni Vigna
Department of Computer Science
University of California, Santa Barbara, USA
{marco,rusvika,nomed,vigna}@cs.ucsb.edu

Abstract

In the last few years, several approaches have been proposed to perform vulnerability analysis of applications written in high-level languages. However, little has been done to automatically identify security-relevant flaws in binary code.

In this paper, we present a novel approach to the identification of vulnerabilities in x86 executables in ELF binary format. Our approach is based on static analysis and symbolic execution techniques. We implemented our approach in a proof-of-concept tool and used it to detect taint-style vulnerabilities in binary code. The results of our evaluation show that our approach is both practical and effective.
Keywords: *Vulnerability analysis, binary static analysis, symbolic execution, taint analysis.*

1 Introduction

Vulnerability analysis is the process of determining if a system contains flaws that could be leveraged by an attacker to compromise the security of the system or that of the platform the system runs on. In comparison to other approaches to security, such as intrusion detection and prevention, the focus of vulnerability analysis is on identifying and eventually correcting flaws, rather than detecting and blocking attacks that exploit a flaw.

Research on vulnerability analysis has produced a number of different approaches to identify security flaws in an automatic—or semi-automatic—fashion. Some of the most promising approaches are based on static analysis techniques. These techniques predict safe and computable approximations of the set of values, or behaviors, that the application could show at run-time [27].

Static analysis techniques represent an appealing building block for security analysis because they provide results without having to actually run an application, thus avoiding risks linked to the execution of malicious programs. Furthermore, by computing safe approximations of a program's behavior, they generally achieve a low rate of false negatives. In other words, if there is a vulnerability in the application under test, in most cases the analysis is able to find it. Finally, static approaches have no impact on the run-time performance of an application and offer the possibility to detect and correct flaws before its deployment. However, the approximate nature of the results provided by static analysis makes it difficult to eliminate false positives. As a consequence, some of the "vulnerabilities" identified by the analysis might actually correspond to correct code.

A significant part of recent efforts in static vulnerability analysis have been directed towards analyzing applications written in some high-level language, such as C, Java, or PHP [34, 21, 36]. However, the application of static vulnerability analysis to binary code is appealing for a number of reasons. First, it is not always the case that the source code of an application is available. For example, some proprietary applications are distributed in binary form only. Second, even when the source code for compiled languages is available, the "What You See Is Not What You eXecute" principle applies [3]. This principle states that the transformations performed by compilers and optimizer tools may subtly—but significantly—alter the actual behavior

of an application, and, consequently, invalidate the results of the analysis performed at the source code level. For example, statements used to overwrite a buffer containing a password may be considered redundant and, thus, be optimized away. Also, the order of instructions could be altered, thereby invalidating a security-critical sequence of events.

In this paper, we explore the problem of performing vulnerability analysis of binaries by using static analysis techniques. We focus on executables written for the Intel x86 family of processors because this is the most widely-used architecture and it presents several characteristics that make our task more challenging.

We restrict our attention to the common class of flaws that consists of the use of data coming from untrusted sources (*tainted data*) in sensitive operations. As an example, consider the "tainted-data-to-system" problem. The `system()` function, which is provided by the C standard library, executes the command(s) specified in its only parameter by invoking a shell to evaluate that parameter. If an attacker controls the value of the parameter passed to `system()`, she is able to execute arbitrary commands with the privileges of the vulnerable process. The same problem exists with the `popen()` function. This vulnerability is, therefore, especially critical in two scenarios. First, if a SUID application is affected, local users are able to mount local-to-root attacks or to impersonate other users of the system. Second, if a network-accessible application (e.g., a CGI program or a network server) is vulnerable, then remote attackers can gain local access.

Consider, for example, the simple program shown in Figure 1. The application is intended to be used as a CGI program that takes as input the name of a machine, sends it five ping packets, and returns to the user the transmission statistics. The invocation of the `ping` command is done through the `popen()` function without performing any validation of the user input. Therefore, if an attacker invokes the program with the input string "`;cat /etc/passwd`", she will have access to the content of the password file [1].

In this paper we make the following contributions to the problem of detecting vulnerabilities in binary programs through static analysis:

- We propose a novel adaptation of context- and path-sensitive symbolic execution to detect the class of vulnerabilities consisting of the use of tainted data in sensitive operations.

- We present a set of techniques and heuristics that makes the analysis of x86 binaries more efficient and practical.

- We implemented a proof-of-concept tool that uses these techniques to analyze real-world binaries.

- We present experimental results showing that our approach is both effective and practical.

[1] A similar vulnerable ASP program was actually present in early releases of the Linksys WRT54G wireless router.

```
int main(int argc, char **argv) {
  char *site, *query, cmd[128];
  FILE *f;

  query = getenv("QUERY_STRING");
  if ((query &&
      (site = strchr(query, '=')))) {
    site++;
  } else {
    site = "localhost";
  }
  snprintf(cmd, sizeof(cmd),
           "ping -c 1 %s", site);
  f = popen(cmd, "r");
  ...
```

Figure 1. Example of a vulnerable program.

We wish to emphasize that the goal of this paper is to present a vulnerability detection technique, as opposed to a verification technique. Our approach is neither sound nor complete, that is, it is possible that programs are flagged as vulnerable when they are indeed correct and that actual vulnerabilities are not recognized. As we will discuss, the main sources of imprecision in the analysis are the handling of loops, the modeling of the x86 architecture and instruction set, and the modeling of `libc` functions.

The rest of the paper is organized as follows. In Section 2 we discuss work related to ours. In Section 3, we present the static analysis techniques and models that we

use to analyze x86 binaries. Section 4 contains a discussion of how such techniques and models are leveraged to detect taint-style vulnerabilities. The results from experiments with the prototype tool that we implemented are presented in Section 5. Finally, Section 6 concludes and indicates future work.

2 Related Work

As mentioned earlier, a significant part of recent efforts in static vulnerability analysis have been directed towards analyzing applications written in some high-level language. Due to the lack of space and the significant amount of work done in this direction, we are not going to discuss this work here. An interested reader can refer to some of the references provided in Section 1.

The problem of identifying vulnerabilities in binary code, on the other hand, has been mostly tackled using dynamic techniques. Within this class of techniques it is possible to distinguish between testing-based and monitoring-based approaches. Testing-based approaches try to trigger a vulnerability by exercising an application with random or malicious inputs, [22, 28]. Monitoring-based approaches, instead, examine the execution of an application during normal use, looking for anomalous behaviors, [24]. In particular, a whole area of research has focused on ways to detect attacks on the basis of the analysis of the system call invocations performed by a program [10, 12, 23].

The dynamic analysis approach described in [26] is more closely related to ours. It describes TaintCheck, a dynamic taint analysis tool, which can detect overwrite attacks (e.g., format string attacks, buffer overflows) on x86 binaries. TaintCheck differs from our approach in a number of ways, mainly due to fundamental differences between static and dynamic analysis approaches. For instance, since TaintCheck performs its analysis on a running program, it covers only those execution paths that are traversed during a given execution. Our tool achieves a more complete code coverage since static analysis is usually applied on all possible inputs and paths. Second, TaintCheck detects attacks at runtime, while our tool finds vulnerabilities without the need of running the analyzed application. Finally, the dynamic monitoring performed by TaintCheck may cause a significant degradation of the performance of an analyzed program. In our tool all analyses are static and, thus, have no influence on the performance of an application. Our tool, on the other hand, suffers from a higher false positive rate due to inherent limitations of static analysis.

While the use of dynamic techniques has been proven useful for the detection of vulnerabilities and their exploitation, static techniques provide a set of advantages that make them appealing. Unfortunately, there are a number of challenges that have to be overcome by static binary analysis tools [30], and, as a consequence, there are only few existing approaches to the static detection of vulnerabilities in binary programs.

In [5], the authors describe a static method to generate attack-independent signatures for vulnerabilities in binary code. Their approach is complementary to ours in that once a vulnerability is known, they generate a signature that detects attack attempts. Instead, we focus on the problem of identifying vulnerabilities.

Static binary analysis techniques are applied to the detection of malicious code. In [4], the authors describe an approach to statically detecting malicious code in executable programs by abstracting the program into a graph of critical API calls, which is then checked against a policy automaton to determine if it may cause a violation.

Static approaches have also been applied to viruses and worms detection, [7, 8], as well as polymorphic worms detection, [25, 19]. Static analysis has also been applied to rootkit detection [20] and to identifying spyware-like behavior [17]. Our technique is similar to these approaches to malicious code detection in that it statically extracts information from an executable. However, we are concerned with the presence of possible vulnerabilities and not with the detection of malicious code or behavior. Therefore, we use the results of the static analysis in a different manner.

Our work further relates to general techniques of static analysis of binary code. This is a very active area of research and it would be impossible to mention all of the relevant work here. Thus, we reference the results that we leveraged in our work when we discuss specific techniques.

3 Static Analysis of Binaries

Our vulnerability analysis process is logically divided in two phases. We first use several techniques to statically approximate the state of an application dur-

ing execution. Then, we leverage this information to detect vulnerabilities. In this section, we describe the key characteristics of our static analysis approach, which is an extension of the analysis presented in [18]. Our approach has been implemented in a proof-of-concept tool whose architecture is presented in Figure 2.

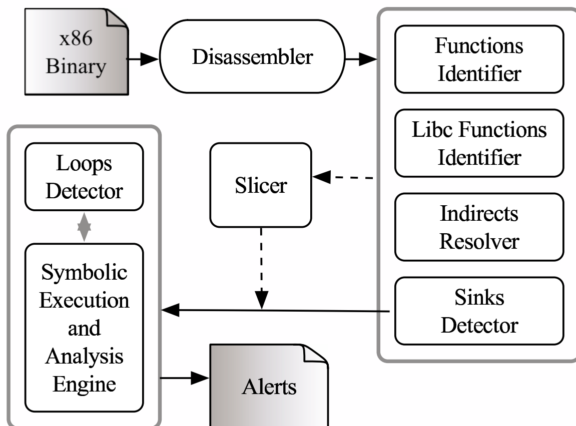

Figure 2. The architecture of our vulnerability detection tool.

Throughout the paper we will use the running example given in Figure 1. Note that the example is in a C-like language for the sake of clarity and simplicity, while our tool operates on binary code. In particular, we assume that our tool will be operating on dynamically-linked x86 executable objects, formatted according to the Executable and Linking Format (ELF). We also assume that the analyzed executable follows a "standard compilation model" [29]: the executables has procedures, a global data region, a heap and a runtime stack; global variables are located at a fixed memory location; local variables of a procedure are stored at a fixed location in the frame stack of that procedure; the program follows the `cdecl` calling convention, and is not self-modifying.

3.1 Prerequisites

In order to perform our analysis, we must first disassemble the binary file and build data structures that enable us to navigate the resulting assembly code. Note that we are able to operate on stripped binaries, i.e., executables that lack debugging and other support information. We use several of the suggestions presented in [13] to deal with the peculiarities of stripped code.

First, the binary program is disassembled using an enhanced version of the basic linear-sweep algorithm as described in [31], thus providing us with an assembly level representation of the program. Note that this version of the linear-sweep algorithm is resilient to the insertion of jump tables corresponding to `switch` statements that might have been inserted into the instruction stream.

Second, we attempt to resolve indirect `call` and `jump` instructions. Resolution of the possible target addresses helps in the identification of functions and the derivation of a complete Control Flow Graph (CFG). We implemented a series of heuristics to determine the target values of indirect branches in some common cases. First, to resolve jump-table-based branches, we backtrack in the code until we reach the instructions that set up the jump table access, thus recovering the base location and the number of entries of the table. Our implementation of this method is compiler-dependent and has proven to work well in practice. Second, some indirect branches are resolved by performing a form of intraprocedural constant propagation. More precisely, we symbolically execute the current function to determine a set of possible targets. Finally, we apply a similar mechanism interprocedurally to resolve indirect branches that derive, for example, from the presence of functions that return function pointers.

Third, it is necessary to identify loops and recursive function calls. To detect loops we use the algorithm described in [32], which, unlike the classical Tarjan's interval-finding algorithm [33], is also able to identify irreducible loops (i.e., loops with multiple entry points). Contrary to popular belief, these appear frequently in optimized binary code. Recursive function calls are identified by applying a standard topological sort algorithm on the function call graph of the program.

Finally, we need to resolve the name of library functions used in a program in order to correctly model execution and identify exit points in the CFG. Since we assume we are working with dynamically-linked ELF binaries, we extract library function names by combining information contained in the Procedure Linkage Table (PLT) and the relocation table of the binary.

3.2 Symbolic Execution

Our analysis technique is based on *symbolic execution* [16]. Symbolic execution consists of interpretatively executing a program by supplying symbols representing arbitrary values instead of concrete inputs, e.g., strings or numbers. The execution is then performed as in a concrete execution, except that the values processed by the program can be symbolic expressions over the input symbols. By doing this, the symbolic execution approximates all possible concrete executions.

Execution State We extended the concrete execution semantics of the x86 assembly language to define the effect of instructions on the *execution state* of a program. In the current implementation of our approach, the execution state models the content of processor registers (the general registers `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, and the stack registers `esp`, `ebp`) and memory locations (both stack and heap). Memory locations and registers can hold both concrete values and symbolic expressions. In the initial execution state, the program counter contains the address of the first instruction of the `main` procedure, the stack registers are initialized to a fixed, concrete value, and all the remaining modeled registers and memory locations are assigned symbolic values.

Symbolic values are introduced in the execution as the result of reading from registers or memory locations that have not been assigned a concrete value or after invoking library functions that read external input, e.g., from files, sockets, and the process environment. In the example of Figure 1, the call to the `getenv()` function introduces a new symbolic value, say v_1, which will be assigned to the register `eax`.

There are a number of limitations in our symbolic executions. In the current implementation of our approach, all symbolic values are assumed to represent integers. We further constrain the expressions to be linear combinations of symbols. Instructions whose effect cannot be modeled as a linear constraint, e.g., multiplication between symbolic values, produce as a result the special symbol `unknown`, which is used to denote that nothing can be asserted about the content of the affected register or memory location.

Branches and Loops While the model described so far would be sufficient to symbolically execute linear code, the presence of branch instructions requires the extension of the execution state with *path conditions*. Path conditions are Boolean expressions over the symbolic input values used as parameters in branching instructions. More precisely, path conditions represent the constraints that the symbolic values must satisfy for an execution to explore the associated path.

In the initial execution state, the set of path conditions is empty. When a branch instruction is executed, both its Boolean condition—in general, a symbolic expression—and its negation are evaluated. When a condition is determined to be feasible, it is added to the current set of path conditions and the execution is "forked" to continue along the feasible branch.

In our running example, when the execution reaches the branch instruction corresponding to the `if` statement, the symbolic value v_1 returned by `getenv()` is checked to determine if it is equal to 0. Since the set of path conditions is empty, both the "true" and "false" branches are determined to be feasible. Therefore, the execution is forked, and continues along the "true" branch with the condition $v_1 \neq 0$ and along the "false" branch with the condition $v_1 = 0$.

To check the feasibility of path conditions, the analysis has to solve systems of linear expressions. Our current prototype uses the Parma Polyhedra Library [1] as its constraint solver.

The symbolic execution algorithm described so far would terminate only for programs that contain loops whose termination condition can be statically determined. Unfortunately, this is not often the case in practice. To handle such loops, we then use a simple heuristic: an execution can visit the same loop no more than three times. At the fourth visit, the execution is interrupted, an approximation of the effects of an arbitrary number of iterations of the loop is computed, and, finally, the execution is restarted using the approximated state. In practice, all registers and memory locations that were modified in the loop body are assigned the `unknown` symbol.

Finally, recursive function calls are terminated immediately, that is, recursive loops are explored only once.

Alias Analysis Two different expressions are said to be aliases if they point to the same concrete memory location. Clearly, write operations on the location represented by an expression should be reflected on all aliased

expressions. Unfortunately, the problem of determining alias relationships is still open even in high-level languages [14], and is only exacerbated at the binary code level. The current prototype takes the optimistic approach that different expressions refer to different memory locations. This is supported by the fact that most compiler-generated code does not in fact use aliased expressions to point to the same location. A more sophisticated approach could leverage the techniques proposed in [2, 9] to guarantee sound results without sacrificing precision, at the cost, however, of a more complex and expensive analysis.

A related problem is that of write operations on `unknown` memory locations. Potentially, any memory location could be affected by such an operation. Again, we take the optimistic approach and assume that no aliasing occurs, i.e., the write operation only affects a new memory location.

To summarize, the ideal goal of symbolic execution is to traverse all feasible program execution paths. In practice, the set of traversed paths will both include infeasible paths—e.g., paths for which the constraint solver cannot statically determine if the associated condition is infeasible— and miss feasible paths—e.g., loops that are not symbolically executed as many times as in a concrete execution. Nonetheless, symbolic execution is valuable in that it provides a framework to implement path-sensitive and context-sensitive interprocedural analysis. The precision allowed by symbolic execution comes at the expense of speed of analysis. We defer to the next section a discussion of some techniques that we use to control this problem.

4 Vulnerability Analysis

Our approach uses the information extracted by performing symbolic execution to identify vulnerabilities in binary code. In particular, we show how taint analysis can be used to statically detect the use of untrusted data in sensitive operations. In the current implementation of our approach, we focus on identifying insecure uses of the standard C library functions `system()` and `popen()`. Both functions are infamous for being insecure if not used carefully. The problem is that they invoke a subshell to execute a command supplied to them as a string parameter. Thus, if the parameter originates from an untrusted source and is not carefully sanitized, `system()` and `popen()` can be used to execute arbitrary commands.

Despite their issues and constant abuse, `system()` and `popen()` are still widely used. Unfortunately, insecure uses of these functions are still common. For example, an insecure use of `system` was recently found in the `scp` program of the OpenSSH and Dropbear application suites [6].

4.1 Taint Analysis

We approach the problem of identifying insecure uses of the `system()` and `popen()` functions as a general *taint analysis* problem. In taint analysis, data originating from sources outside of the program's control is considered untrusted and is marked as tainted. Then, the propagation of tainted data through the program is traced to check whether it can reach security-critical program points.

Our analysis follows the standard approach to taint analysis and can be conceptually divided into four main parts:

1) identification of *sources* of untrusted data, such as command line parameters, environment variables, data read from files, etc.;

2) identification of sensitive *sinks* (e.g., calls to the `system()` and `popen()` functions);

3) propagation of tainted data;

4) generation of alerts when tainted data reaches a sensitive sink.

In the current prototype, sources of untrusted data are identified using various methods. For example, command line parameters are identified by locating the parameters passed to the `main` function. We consider a parameter of a function to be any memory location that, within that function, is accessed through a positive displacement from the `ebp` register. Environment variables are determined by tracking calls to the `getenv()` function provided by the standard C library. Similarly, other untrusted data, such as the data read from the standard input, files, or sockets, is identified by tracking calls to specific library functions, such as `read()`.

In our analysis, we are considering only two sensitive sinks: the library functions `system()` and `popen()`. After library functions have been identified by analyzing the PLT and the relocation table of a dynamically-linked binary, any call to the `system()` or `popen()` func-

tions can be easily traced. Our tool then raises an alert if any tainted data is used as a parameter to these functions.

Propagation of tainted data from sources to sinks is done by using the symbolic execution technique described in Section 3.2. In particular, taint propagation rules determine the effect of each instruction with regard to the taintedness of its operands. For example, whenever a `mov` instruction is executed with a tainted source operand, the corresponding destination operand also becomes tainted.

Similarly, we model how library functions propagate taint information; in particular, we specify which parameters become tainted after their invocation and whether their return value is tainted. To identify parameters passed to library functions we assume that the binary uses the `cdecl` calling convention. Then, if the modeled function accepts a well-defined number of parameters, we simply read them from the execution stack. Instead, if the function is variadic and accepts a variable number of parameters, e.g., `snprintf()`, the analysis is more complex and relies on a number of simplifying assumptions. First, we assume that the instructions that move parameters on the stack are all in the same basic block, i.e., they are not intermixed with branch instructions. Second, we assume that no parameter is preserved on the stack between two consequent library function calls, even if the two calls share all or some of the parameters. These assumptions hold in our standard compilation model.

In the current implementation, we have models for the most frequently used `libc` functions that may propagate taint information. We taint data read from a process' environment (`getenv()`), from files (e.g., `read()`, `fread()`, `fgets()`) and from network sockets (e.g., `recv()`, `recvfrom()`). Since we consider all sources of external data as untrusted, our analysis is very conservative, thus limiting the possibility of incurring in false negatives. We also model a number of functions that can transfer taint information, such as `strdup()` and `snprintf()`. All other library functions are modeled as empty functions that immediately return after performing the standard function epilogue.

In the example of Figure 1, the analysis identifies the `call` instruction corresponding to the invocation of `getenv()` as a source and the `call` instruction corresponding to the invocation of `popen()` as a sink. The return value of `getenv()`, v_1, is tainted. We have seen that the symbolic execution proceeds along three paths. Along two of these paths the value corresponding to the variable `site`, v_2, has a concrete value (the address of the string `localhost`) and is not tainted. On the remaining path, however, taint is propagated from v_1 to v_2. The call to `snprintf()` simply propagates the taintedness status of its parameters to the symbolic value associated with the destination buffer, say v_3. Finally, v_3 is passed as a parameter to the `popen()` function. During execution of the third path, along which v_3 is tainted, our tool will raise one alert. The other two executions do not generate any alarm.

The `libc` library does not provide any well-defined functions to sanitize user input. From our experience with C programs, in general, sanitization is done by developer-written routines that iteratively parse strings looking for suspicious characters. These routines are often ad-hoc and error-prone. As a consequence, in our current implementation we take a conservative approach and do not model untainting.

4.2 Optimizations

The symbolic execution performed by our tool explores all possible execution paths in a program. When performing taint analysis, however, we are only interested in those paths through which a sensitive sink can be reached. To this end, examining the remaining paths would not provide any benefit. To avoid this problem we implemented a simple form of program slicing.

Program slicing is a well-known technique that finds all statements in a program which might affect the value of a given variable at a specific program point [35]. In general, the problem of computing a minimal program slice is undecidable. However, there are sophisticated methods that are able to compute highly precise slices [15]. For our purposes, precision is not the main issue. Instead, we simply want to conservatively reduce the number of paths that are explored during taint analysis.

Therefore, starting from the function `main()`, we find all instructions in the binary that are on a path to a sink (e.g., calls to `system()` or `popen()`). We do this by first constructing the interprocedural CFG of the analyzed program and subsequently traversing it in a context-sensitive manner starting from all identified sinks, terminating when the beginning of `main()` has been reached. Even though this technique is not able to

Application	Size (KB)	Basic Blocks	Basic Blocks after Slicing	Detection Time (sec)	Code Coverage
scp 0.47	46.3	1023	599	1	62%
a2ps 4.13	570.6	6403	3499	62	37.4%
ppmcolors 10.33.0	12.0	41	20	1	100%
irexec 0.7.2	7.0	88	36	0.01	100%
autrace 1.0.14	57.7	315	158	-	93%

Table 1. Test Results.

obtain highly precise program slices, in many cases it prunes execution paths that are not relevant to our analysis.

This approach, however, has several limitations. A fundamental precondition of our slicing technique is that the complete interprocedural CFG for the program be available. The presence of indirect jumps or calls makes it harder to satisfy this requirement. While, in general, it is impossible to statically resolve all indirect transfers of control, we implemented some heuristics that handle the most common cases (see Section 3.1 for more details). If some of the indirect jumps or calls are not resolved, no slicing is done and the whole program is analyzed.

5 Evaluation

We performed a series of experiments with our prototype tool to evaluate its ability to detect taint-style vulnerabilities. The test dataset includes two applications with known taint-style vulnerabilities (Dropbear scp [6] and GNU a2ps [11]) and a set of utilities that use the `system()` and `popen()` calls. All experiments were run on a Fedora Core 4 system, equipped with a 3.60 GHz Pentium 4 processor and 2GB RAM. All executables were obtained using the gcc 4.0.0 compiler using its standard configuration and the optimization level `O2`.

Experimental results are shown in Table 1. The table reports the size of the binaries in KB and in terms of the number of basic blocks identified by our tool before and after performing simple slicing. The detection time indicates the time required to identify all code points where tainted data is used in `system()` or `popen()` functions. The coverage column reports the percentage of basic blocks visited at least once during the first 30 minutes of execution. The detection performance results are reported in Table 2.

When applied to the presented test set, our tool identifies all uses of tainted data in sensitive operations, i.e., the false negatives rate is 0%. The tool generates one false positive in `scp`, `ppmcolors`, and `irexec`. False positives occur for different reasons. First, our tool raises an alarm when user-provided input is, by design, allowed to reach the `system()` or `popen()` functions. Such alarms can be useful to raise the attention of an analyst or systems administrator on binaries whose security could be compromised if their intended use changed (e.g., if they were made accessible to remote, untrusted users). This is the case of the false positive generated when analyzing `scp` and `irexec`. Second, our tool does not include any form of type inference analysis and, consequently, generates an alert also when the type of the input prevents a successful attack. In `ppmcolors`, for example, user input is first converted to an integer and then used to compose the command string passed to `system()`. In this case, an attacker would not be able to inject a malicious command in the application. Our tool correctly identifies `autrace` as containing no vulnerabilities because the `system()` function used in this application does not utilize any user-provided input.

The symbolic execution technique that underlies our analysis reconstructs all feasible execution paths in a program. However, the number of distinct paths increases roughly exponentially with the number of instructions and, thus, can be extremely large even in small programs. Except for the most trivial cases, an exhaustive analysis would then require extremely long execution time. While this is acceptable in some contexts, e.g., verification, we believe that a more practical approach is needed for our detection purposes. Therefore, we ran our experiments setting a timeout of thirty minutes and, in our results, we report only vulnerabilities

Application	Number of Calls			False Positives	False Negatives
	Total	Vulnerable	Detected		
scp	2	1	2	1	0
a2ps	2	2	2	0	0
ppmcolors	1	0	1	1	0
irexec	1	0	1	1	0
autrace	2	0	0	0	0

Table 2. Analysis Results.

detected during this interval.

In the current prototype of our tool, we have implemented several strategies to explore execution paths achieving good path coverage and low memory requirements. The depth-first search strategy (which consists of always resuming the last unexplored branch) is optimal in terms of memory usage, but can spend significant time completely exploring uninteresting paths. The random selection of unexplored branches has a higher impact on memory, but covers the search space evenly. In the reported tests we used a third strategy that combines depth-first and breadth-first search: the execution proceeds in a depth-first fashion for a given interval of time. When the time interval elapses, execution resumes from the branch corresponding to the least executed basic block in the program and continues in depth-first mode.

6 Conclusions

We have presented a novel adaptation of binary analysis techniques to statically detect vulnerabilities in x86 executables. We have described a number of techniques and heuristics that we use to perform this analysis in practical cases. We have implemented our approach in a proof-of-concept tool and evaluated its performances on a number of real-world programs. The results of our tests show that the approach is practical and achieves good detection performances.

In the future, we plan to extend our approach in different directions. First, we intend to include more sophisticated analysis, e.g., to better model memory accesses and the abstraction of loops. Second, we want to explore the idea of complementing static analysis with dynamic analysis to detect vulnerabilities in executables.

7 Acknowledgments

We would like to thank Christopher Kruegel and William Robertson for providing us with the symbolic execution engine and useful suggestions throughout this project.

This research was supported by the Army Research Office, under agreement DAAD19-01-1-0484, and by the National Science Foundation, under grants CNS-0209065, CCR-0238492 and CCR-0524853.

References

[1] R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly Not Closed Convex Polyhedra and the Parma Polyhedra Library. In *Proc. of the 9th Intl. Static Analysis Symp.*, pages 213–229, 2002.

[2] G. Balakrishnan and T. Reps. Analyzing Memory Accesses in x86 Executables. In *Proc. of the 13th Intl. Conf. on Compiler Construction*, pages 5–23, 2004.

[3] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. WYSINWYX: What You See Is Not What You eXecute. In *Proc. of the IFIP Working Conf. on Verified Software: Theories, Tools, Experiments*, 2005.

[4] J. Bergeron, M. Debbabi, J. Desharnais, M. M. Erhioui, Y. Lavoie, and N. Tawbi. Static Detection of Malicious Code in Executable Programs. *Int. J. of Req. Eng.*, 2001.

[5] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards Automatic Generation of Vulnerability-Based Signatures. In *Proc. of the 2006 IEEE Symp. on Security and Privacy*, pages 2–16, 2006.

[6] BugTraq. OpenSSH, Dropbear: Insecure use of system() call. http://seclists.org/lists/bugtraq/2006/Feb/0401.html, Feb 2006.

[7] M. Christodorescu and S. Jha. Static Analysis of Executables to Detect Malicious Patterns. In *Proc. of the 12th USENIX Security Symp.*, 2003.

[8] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-Aware Malware Detection. In *Proc. of the 2005 IEEE Symp. on Security and Privacy*, pages 32–46, 2005.

[9] S. Debray, R. Muth, and M. Weippert. Alias Analysis of Executable Code. In *Proc. of the 25th ACM SIGPLAN-SIGACT Symp. on Principles Of Programming Languages*, pages 12–24, 1998.

[10] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A Sense of Self for UNIX Processes. In *Proc. of the 1996 IEEE Symp. on Security and Privacy*, pages 120–128, 1996.

[11] Full-Disclosure. a2ps executing shell commands from file name. `http://archives.neohapsis.com/archives/fulldisclosure/2004-08/1026.html`, Aug 2004.

[12] J. T. Giffin, S. Jha, and B. P. Miller. Efficient Context-Sensitive Intrusion Detection. In *Proc. of the 11th Network and Distributed System Security Symp.*, 2004.

[13] L. C. Harris and B. P. Miller. Practical Analysis of Stripped Binary Code. In *Proc. of the 2005 Workshop on Binary Instrumentation and Applications*, 2005.

[14] M. Hind. Pointer Analysis: Haven't We Solved This Problem Yet? In *Proc. of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools Engineering*, pages 54–61, 2001.

[15] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. In *Proc. of the ACM SIGPLAN 1988 Conf. on Programming Language Design and Implementation*, pages 35–46, 1988.

[16] J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[17] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer. Behavior-based spyware detection. In *Proc. of the 15th USENIX Security Symp.*, 2006.

[18] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating Mimicry Attacks Using Static Binary Analysis. In *Proc. of the 14th USENIX Security Symp.*, pages 161–176, 2005.

[19] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic Worm Detection Using Structural Information of Executables. In *Proc. of the 8th Intl. Symp. on Recent Advances in Intrusion Detection*, pages 207–226, 2005.

[20] C. Kruegel, W. Robertson, and G. Vigna. Detecting Kernel-Level Rootkits Through Binary Analysis. In *Proc. of the Annual Computer Security Applications Conf. (ACSAC)*, pages 91–100, Tucson, AZ, December 2004.

[21] B. Livshits and M. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proc. of the 14th USENIX Security Symp.*, pages 271–286, 2005.

[22] B. P. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[23] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel. Anomalous System Call Detection. *ACM Transactions on Information and System Security*, 9(1), Feb 2006.

[24] N. Nethercote and J. Fitzhardinge. Bounds-Checking Entire Programs Without Recompiling. In *Informal Proc. of the 2nd Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management*, 2004.

[25] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *Proc. of the 2005 IEEE Symp. on Security and Privacy*, pages 226–241, 2005.

[26] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proc. of the 12th Annual Network and Distributed System Security Symp. (NDSS'05)*, 2005.

[27] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[28] Oulu University Secure Programming Group. PROTOS Test-Suite: c06-snmpv1. Technical report, University of Oulu, Electrical and Information Engineering, 2002.

[29] T. Reps, G. Balakrishnan, and J. Lim. Intermediate-Representation Recovery from Low-Level Code. In *Proc. of the 2006 ACM SIGPLAN Symp. on Partial Evaluation and semantics-based Program Manipulation*, pages 100–111. ACM Press, 2006.

[30] T. Reps, G. Balakrishnan, J. Lim, and T. Teitelbaum. Next-generation platform for analyzing executables. In *Proc. of the 3rd Asian Symp. on Programming Languages and Systems*, pages 212–229, 2005.

[31] B. Schwarz, S. Debray, and G. Andrews. Disassembly of Executable Code Revisited. In *Proc. of the 9th Working Conf. on Reverse Engineering*, pages 45–54, 2002.

[32] V. C. Sreedhar, G. R. Gao, and Y.-F. Lee. Identifying Loops Using DJ Graphs. *ACM Transactions on Programming Languages and Systems*, 18(6):649–658, November 1996.

[33] R. E. Tarjan. Testing Flow Graph Reducibility. *Journal of Computer and System Sciences*, 9(3):355–365, 1974.

[34] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proc. of the Annual Network and Distributed System Security Symp. (NDSS 2000)*, 2000.

[35] M. Weiser. Program Slicing. In *Proc. of the 5th Intl. Conf. on Software Engineering*, pages 439–449, 1981.

[36] Y. Xie and A. Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *Proc. of the 15th USENIX Security Symp.*, 2006.