TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting

Winnie Cheng[1], Qin Zhao[2], Bei Yu[2] and Scott Hiroshige[1]
[1]Massachusetts Institute of Technology
[2]Singapore-MIT Alliance
32 Vassar Street, Cambridge, MA 02139 USA
{wwcheng, zhaoqin, beiyu, skhirosh}@mit.edu

Abstract

TaintTrace is a high performance flow tracing tool that protects systems against security exploits. It is based on dynamic execution binary rewriting empowering our tool with fine-grained monitoring of system activities such as the tracking of the usage and propagation of data originated from the network. The challenge lies in minimizing the run-time overhead of the tool. TaintTrace uses a number of techniques such as direct memory mapping to optimize performance. In this paper, we demonstrate that TaintTrace is effective in protecting against various attacks while maintaining a modest slowdown of 5.5 times, offering significant improvements over similar tools.

1 Introduction

Critical vulnerabilities and security exploits are the norms in today's computer systems. While the Internet has enhanced our communication, it is a primarily unregulated infrastructure where users are susceptible to malicious attacks. Worms like CodeRed and CodeRedII are capable of spreading to thousands of victims within minutes[14]. Signature-based scanning is often too slow to respond to these attacks. Hence, a protection mechanism should provide immunity to known as well as unknown attacks.

Dynamic taint tracing has been proposed to counter exploits from most critical vulnerabilities. This technique keeps track of the propagation of untrusted (tainted) data during program execution. Tainted data may represent any untrusted source such as user input, packets from a network socket, or data read from specific files/devices. Taint tracing is based on a program's dynamic behavior without the need of *a priori* knowledge of a signature, and is therefore effective even against future attacks.

In a summary of vulnerabilities in the Red Hat operating system, buffer overflow and overwrite attacks were identified as the dominant culprits [1]. Attacks on these vulnerabilities can be easily avoided with data flow tracing. For example, a buffer overflow vulnerability in ATPhttpd [11] allows an attacker to overwrite the return address of a function when a maliciously-crafted string is entered as the file name. The attacker can then direct program execution to gain control of the host machine. However, if the execution is monitored using dynamic taint tracing, the use of tainted data as the return target can easily be detected.

Currently, there are three ways to track taint information on data:

Interpreter-based approach: Perl, an interpreted language, provides a taint mode to keep track of untrusted data [2].

Architecture-based approach: [15] and [7] monitor taint information propagation by adding architectural features to processors.

Instrumentation-based approach: Different from the previous two approaches, TaintCheck[10] inserts additional code into original application to trace and maintain information about the propagation.

However, these approaches have various drawbacks. For instance, interpreter-based approaches can protect only against vulnerabilities in language-specific code. An architecture-based approach requires custom hardware support. Furthermore, hardware approach makes it difficult for system administrators to tailor security policies for individual software applications. Instrumentation-based approach suffers from significant performance overhead prohibiting their use in real-time applications. For example, TaintCheck [10] demonstrated a slowdown of over 30 times when compared against native execution. The overhead is primarly due to the way taint information on data is maintained and propagated.

TaintTrace uses an instrumentation approach to dynamically trace the propagation of taint data. It is based on DynamoRIO [5] and consists of a number of optimizations to

Proceedings of the 11th IEEE Symposium on Computers and Communications (ISCC'06)
0-7695-2588-1/06 $20.00 © 2006 IEEE

keep the overhead low. It is able to protect against a broad set of exploits such as format string and buffer overflow.

Our approach has the following attractive properties:

- **Language independence.** Our tool operates at the binary level and can be used for applications written in any programming language. Moreover, no source code modification or recompilation is needed. This is especially useful in protecting legacy software from being attacked.
- **Comprehensive tracing.** Our instrumentation can be performed on all binary code in user mode, hence our tool can trace data during the execution of the application code as well as of all shared libraries.
- **Real-time usage.** By applying optimizations like direct mapping, our tool is the first of its class to achieve acceptable performance for practical use.

The remainder of the paper is organized as follows. Section 2 gives an overview of our system. Section 3 describes the details of our design decisions and implementation techniques. Experimental results are discussed in Section 4. Section 5 gives an overview of related work and previous attempts at the problem. Finally, Section 6 concludes our work and addresses potential issues to be explored as future work.

2 System Overview

Our system consists of four components. A *configuration file* is used to specify the security policy. The *shadow memory* is a data structure used to maintain the taint information of application data. *Program monitor* is the core module used to perform the instrumentation, intercept system calls, and enforce security policies. A customized *loader* is used to load the application binary, shadow memory, and program monitor into different memory spaces.

To start an application, our loader first loads the various components into specific memory spaces and then passes control to the program monitor. The program monitor reads the configuration file and sets up the tracing policy. It also initializes the shadow memory, that is, it marks the untrusted sources specified by the configuration file as tainted, and other sources as clear.

After initialization, the application executes under our program monitor. All the code to be executed in user mode is first copied into the code cache. This includes application code and shared libraries. The program monitor inserts additional code for maintaining, propagating, and checking taint status before executing the code. In this way, we achieve comprehensive information flow tracing. At critical program points specified by our policy (e.g. indirect branch), run-time condition checking is performed to restrict sensitive data usage.

3 Design and Implementation

3.1 Policy

A configuration file is used to specify security policies enforced by the program monitor. It is write-protected, restricting modification privileges to system administrators. The administrator can define data origins that should be marked as untrusted. The origins may be network sockets, specific input devices, or memory locations. In addition, administrator can activate a set of tracing policies and enable certain alerts.

We define the *residence* of tainted data to be a a memory location or a register. The taint tracing policy defines how information flow is tracked as tainted data propagates among different residences. This can happen in four ways:

1. *Copy Propagation*: Tainted data is copied from one residence to another residence.
2. *Arithmetic Propagation*: The tainted data is the input operand of a mathematical or logical transformation.
3. *Address Propagation*: Tainted data can be used to calculate a memory address and hence, can propagate sensitive data through a table lookup approach.
4. *Control Propagation*: Tainted data may also be propagated through deliberate control transfer. For instance, code like `if(x == 1) y = 1; else if(x == 2) y = 2; ...` uses tainted data `x` to influence the value of `y`.

Our default tracing policy covers propagation types 1 and 2. It maintains the invariant: *the output (destination) data is tagged as tainted if and only if any of the input (source) data is tainted*. This default policy does not trace propagation type 3 because it is a rare source of attack in practice. However, our tool can easily be extended to provide this as an option. Naive taint propagation of control-dependent data (type 4) can lead to a large number of false positives. This is a fundamental problem faced by previous approaches [10, 6]. Instead, our tool allows users to specify critical places in their program where control flow should not be influenced by tainted data.

3.1.1 Taint tracing

Similar to Memcheck [12], we associate each general purpose register and each byte of memory with a shadow memory byte to maintain its taint status: 1 represents tainted and 0 represents untainted. This information is propagated as instructions are executed.

3.1.2 Performance Challenges

Although the concept of taint tracing is simple, a similar approach described in [10] reports a slowdown of greater than 30 times when compared against native execution. Minimizing the overhead is a challenging task.

There are two types of overhead. One is *instrumentation overhead*. This is the overhead caused by performing the code instrumentation. The other is *tracing overhead*. This is the overhead incurred by executing the instrumented code to propagate taint status. We expect the tracing overhead to be much larger than the instrumentation overhead. This is because instrumentation is performed once for each application instruction while the instrumented code may be executed many times. This agrees with our experimental findings. Therefore, our optimization focuses on the tracing overhead, that is, reducing the number of instrumented instructions for each application instruction.

The tracing overhead can be divided into 3 parts:

1. **Shadow memory mapping.** This is the overhead in mapping operands of application instructions to their shadow memory.
2. **Register spilling.** As a result of the dynamic insertion of flow tracing instructions, some general purpose registers need to be saved and restored to avoid disrupting the intended execution of the application code. For example, if the instrumented code modifies the EFLAGs register, this register must be saved prior to the execution of the instrumented code and the saved value restored later.
3. **Propagation overhead.** This overhead is incurred in order to propagate the taint information as data flow from different residences.

```
l2 = l1[(addr >> 16) & 0xffff];
shadow = &l2[addr & 0xffff];
```

(a) C code for mapping addr to shadow.

```
01. mov addr → eax
02. sar eax, 10h → eax
03. and eax, 0ffffh → eax
04. mov [eax*4+l1] → eax
05. mov eax → l2
06. movzx word addr → eax
07. lea [eax*4+l2] → eax
08. mov eax → shadow
```

(b) Instructions generated by `gcc` for (a).

Figure 1. Shadow memory mapping with page-table-like structure

3.1.3 Optimizations

We apply optimization techniques to reduce all of the above tracing overheads. First, we realize the large mapping overhead of page-table-like shadow memory structure used in [10]. In Figure 1, it takes 8 instructions to locate the shadow memory (shadow) for the operand (addr). We use a simple addressing strategy that maps the shadow memory byte by adding a constant offset, `shadow_base`, to the application memory byte address. Our customized loader partitions the memory space to support this mapping. This byte-to-byte mapping makes taint propagation simple and efficient. Second, we minimize register spilling with two techniques. We use dead registers whenever possible. Also, we check whether an application instruction will overwrite the EFLAGs. If so, we need not save and restore the EFLAGs, for example, in arithmetic instructions.

3.1.4 Implementation

We implemented our tool in Linux on x86 architecture using DynamoRIO to perform instrumentation. As a proof of concept, our tool instruments all instructions except specialized instructions.

Our loader is implemented by modifying the source code of Valgrind 2.4.0 [3]. It consists of two stages. In stage 1, it loads the code of stage 2 into the *monitor space* (`0xb0000000` to `0xbfffffff`) and transfers control to stage 2. In stage 2, the application and its shared libraries are loaded into the *application space* (`0x000000000` to `0x57f00000`). It also loads DynamoRIO into the monitor space and transfers control to DynamoRIO. DynamoRIO loads our program monitor, `dr-instrument.so`, implemented as a shared library, into the monitor space. DynamoRIO constructs basic blocks for execution and `dr-instrument.so` is used to perform the instrumentation and intercept system calls.

We intercept system calls for several purposes: allocating shadow memory, marking taint status for data read from files or socket, and modifying temporary file operations. In Linux, a system call is implemented by the soft interrupt instruction `int80`. The system call ID and its parameters are passed through the general purpose registers. `dr-instrument.so` inserts instructions to call our functions `before_syscall` and `after_syscall` immediately before and after `int80`. Function `before_syscall` examines the system call ID. If the system call is `mmap` or `mmap2` requesting allocation of memory, its parameters are modified to request memory from the application space. Function `after_syscall` examines the result returned from Linux, to check if the memory request is successful. If so, the corresponding shadow memory is also allocated and initialized.

Another task of `dr-instrument.so` is to perform the instrumentation. Due to the complexity of x86 instruction, instrumentation is difficult and tedious. Valgrind [9] first translates x86 instruction to RISC-like Ucode, per-

forms transformation and instrumentation on the Ucode, and then translates Ucode back to x86 instructions. In this way, it is easy to perform code transformation. However, it cannot produce optimal instrumented code. For performance reasons, we perform the instrumentation on the x86 instruction set directly using DynamoRIO. We focus mainly on the following two types of instructions for tracing taint data propagation: data movement instructions (e.g. `mov`, `push`, and `cmov`) and arithmetic instructions (e.g. `add` and `sar`).

The instrumented code for each application instruction performs the following tasks in sequence: (1) spill a few registers for storing taint status, (2) map the original operand (register or memory address) to its shadow memory, (3) load the taint status from shadow memory into the spilled registers, (4) store the status into the destination's shadow memory, and (5) restore the spilled registers. If the instrumented instructions modify the EFLAGs register, we need extra instructions to save and restore the EFLAGs register.

Our optimizations can reduce the number of instructions for (1), (2) and (5) to zero in most situations. Also, the byte-to-byte mapping between application memory and shadow memory simplifies (3) and (4) significantly. Figure 2 and Figure 3 show two examples of our instrumentation for load and store, respectively. Here, the `shadow_base` is 0x57f00000.

▷ load data from [edx+0x8] to eax
– **mov** [edx+0x8] → eax

(a) Before instrumentation.

▷ load taint status from shadow memory of [edx+0x8]
– **mov** [edx+0x57f00008] → eax
▷ store taint status into shadow memory of eax
– **mov** eax → [0xb11f1460]
▷ original instruction
– **mov** [edx+0x8] → eax

(b) After instrumentation.

Figure 2. Instrumentation for load

▷ store data from eax to [edx+0x8]
– **mov** eax → [edx+0x8]

(a) Before instrumentation.

▷ original instruction
– **mov** eax → [edx+0x8]
▷ load taint status from shadow memory of eax
– **mov** [0xb11f1460] → eax
▷ store taint status to shadow memory of [edx+0x8]
– **mov** eax → [edx+0x57f00008]
▷ restore eax
– **mov** [edx+0x8] → eax

(b) After instrumentation.

Figure 3. Instrumentation for store

We implemented four types of taint status checking. First, for the `printf` routine family, we check whether the format string argument is tainted or not. Second, we check the taint status of the jump target, such as a return address or function pointer. This is similar to program shepherding [8]. Third, we check, in the case of indirect jump, whether the data storing the jump target is tainted. Finally, we examine the taint status of some critical policy-specified variables or control points.

3.1.5 Discussion

There is a trade-off between the efficiency of our direct shadow memory mapping and the usage of memory space. Our design reduces the number of instrumented code at the cost of halving the effective application memory usage. Using a page-table-like shadow memory structure as in TaintCheck [10] can keep the shadow memory usage minimal since memory usage for maintaining taint status grows on-demand. Therefore, our tool is more suitable for large servers with sufficient memory and in cases where the application performance is a concern.

With our design, the shadow memory becomes a critical area that must be protected from malicious access. Our shadow memory mapping strategy can prevent attackers from directly modifying the shadow memory as long as the attackers' code is under the control of DynamoRIO. The reason is that when attackers issue an instruction to access an address `addr` in the shadow memory area, the instrumented code will access memory at `addr+shadow_base`, which is beyond the boundary of the shadow memory area. This will cause an invalid memory access exception.

4 Experimental Evaluation

4.1 Effectiveness

We evaluated our taint tracing tool with synthetic exploits based on published vulnerabilities. Here, we present the results for format string, buffer overflow, and critical variable vulnerabilities.

4.1.1 Detecting format string attack

A sample program that mimics a real-world format string attack was written to evaluate TaintTrace's ability to detect the attack. The program accepts a user-supplied format string as the argument to `printf()`. The string has been maliciously crafted to reveal sensitive program data. When the same program was run through our tool, TaintTrace correctly detected that the `printf()` argument was tainted.

4.1.2 Detecting buffer overflow attack

Buffer overflow vulnerabilities allow attackers to write beyond allocated addresses into memory that may be used to store the return address of a function. Our test program copies input from a file to an unchecked local buffer in such a way that the return address is overwritten to point to the sensitive function `grant_access()`. TaintTrace correctly identified the occurrence of the tainted return address. In addition, it successfully detected all the buffer overflow attacks described in [16].

4.1.3 Detecting critical variable attack

We also ran TaintTrace against a malicious program in which a variable that determines the control flow of the program is overwritten by some maliciously devised input, causing the program to grant unauthorized rights to the attacker. Our taint tracing tool successfully determined that the critical variable was tainted.

4.2 Performance

We benchmarked the performance of TaintTrace using a subset of the industry-standard SPEC2000 INT. Our evaluation is done on a system with 2.8GHz Pentium 4 processor, 1024K L2 Cache, 1024MB of RAM, and 2048M swap, running Fedora Core 3.

First, we profiled the distribution of execution time of applications running with our tracing tool. The result shows that more than 95% of time is spent executing application code and instrumented code for almost all benchmarks. This justifies our earlier claim that tracing overhead is the dominant factor.

In order to evaluate the overhead of taint tracing, we ran TaintTrace with all security checks switched off. For each workload, we measured the running time of native execution, execution with our tracing tool, and execution with Valgrind Memcheck. Memcheck uses page-table-like shadow memory and traces status propagation in shadow memory. Figure 4 compares the relative slowdown between Valgrind Memcheck and our tracing tool under different workloads.

As can be observed from the figure, TaintTrace outperforms Valgrind Memcheck greatly on most workloads. The average relative slowdown of our tracing tool over all the workloads is 5.53. It is much smaller than Valgrind Memcheck's average slowdown of 29.62. We did not compare our tracing tool with TaintCheck [10] due to the lack of their source code. However, as reported in their paper, their performance is worse than Valgrind Memcheck.

The slowdown of our tracing tool is below 10 for most workloads. Twolf with test input is the exception because its native execution is already very fast, using only about 0.4s, in which case our tracing tool spent a large portion of time in initialization (about 80% according to our experiment). Under those workloads with ref input, the slowdown of our taint tracing tool is much smaller than those with test input. It is important to note that most of the workloads we tested are CPU-bounded and we expect the overhead of TaintTrace to be even smaller with I/O-bound workloads.

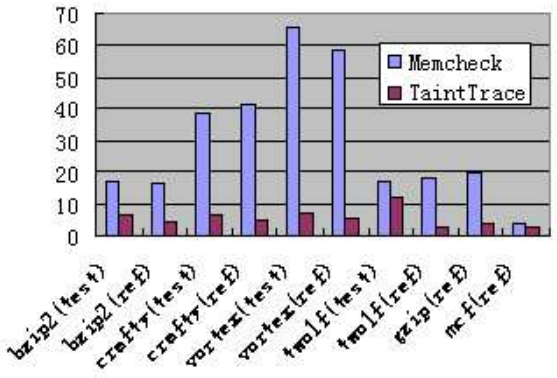

Figure 4. Relative slow down

5 Related Work

5.1 Program Monitoring

Program Shepherding [8] is a run-time monitoring system that keeps track of whether code has been modified since it was loaded, and checks each control transfer to determine if the destination basic block has been modified or not. However, it cannot prevent attacks that exploit vulnerabilities in existing code.

5.2 Taint Tracing and Analysis

One of the most familiar work on data tainting is Perl's taint mode which can prevent both obvious and subtle traps in code execution. While in taint mode, data from potentially untrusted sources, such as network sockets, is tagged as tainted. The interpreter then vigilantly checks for malicious behavior based on these taint tags.

TaintBochs [6], Information Flow Tracking [15] and Minos [7] all perform taint tracing at the hardware level. TaintBochs [6] is built on top of the open source IA-32 simulator Bochs. It is a tool based on whole-system simulation analyzing how sensitive data are handled in large programs. The information flow tracking project [15] designs a hardware mechanism for tracking information flow dynamically. It identifies spurious information flows and restricts

their usage. Minos [7], is a micro-architecture that implements Biba's low-water-mark integrity checking on individual words to detect attacks at run-time.

The main limitation of these three systems is that they require specialized hardware. Without this custom hardware, hardware emulators can be used but performance becomes unacceptable. Moreover, such designs are unable to offer enough flexibility in configuring user-specific security policies.

TaintCheck [10] is the most similar to our work. It instruments code using Valgrind [3]. As mentioned before, its page-table-like structure contributes to its over 30 times slowdown when compared against native execution.

There are also a variety of related work in static program analysis. [13] developed a static analysis system for automatically detecting format string bugs at compile-time. [4] adds system-specific extensions to the compiler and uses tainting-style analysis to find security errors. Due to the lack of run-time information, most of them are overly conservative and inaccurate.

6 Conclusion and Future Work

In this paper, we present TaintTrace, an efficient information flow tracing and program monitoring security system. Our system is able to protect against various forms of attacks including the most widely exploited buffer overflow and format string attacks. Our evaluation demonstrated that TaintTrace is more efficient and practical than similar tools.

Currently, the taint tracing performs instrumentation without knowledge of any global information and hence produces some redundant code. By analyzing the data flow of an entire basic block, we can further minimize the register spilling overhead. This information can also help reduce the overhead on redundant taint propagation. We will be investigating these techniques as future work. Also, it is desirable to provide system logging features and TaintTrace can be extended to log system calls and other run-time information. If an intrusion is detected later, the logged information can help an administrator in analyzing the attack and revealing vulnerabilities.

References

[1] Classification of red hat security alerts 2003. http://www.and.org/vstr/security_problems.html.

[2] Perl security manual page. http://www.perldoc.com/perl5.6/pod/perlsec.html.

[3] Valgrind. http://valgrind.org/.

[4] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, 2002.

[5] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, September 2004. http://www.cag.csail.mit.edu/rio/.

[6] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *13th USENIX Security Symposium*, 2004.

[7] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *MICRO-37*, 2004.

[8] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *11th USENIX Security Symposium*, 2002.

[9] N. Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, University of Cambridge, November 2004.

[10] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *12th Network and Distributed System Security Symposium*, 2005.

[11] Y. Ramin. http://www.redshift.com/ yramin/atp/atphttpd/.

[12] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the USENIX'05 Annual Technical Conference*, 2005.

[13] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *10th USENIX Security Symposium*, 2001.

[14] S. Staniford, V. Paxson, and N. Weaver. How to own the internet in your spare time. In *11th USENIX Security Symposium*, 2002.

[15] G. E. Suh, J. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS-XI*, 2004.

[16] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *10th Network and Distributed System Security Symposium*, 2003.