Understanding Data Lifetime via Whole System Simulation

Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, Mendel Rosenblum
{jchow,blp,talg,kchristo,mendel}@cs.stanford.edu
Stanford University Department of Computer Science

Abstract

Strictly limiting the lifetime (i.e. propagation and duration of exposure) of sensitive data (e.g. passwords) is an important and well accepted practice in secure software development. Unfortunately, there are no current methods available for easily analyzing data lifetime, and very little information available on the quality of today's software with respect to data lifetime.

We describe a system we have developed for analyzing sensitive data lifetime through whole system simulation called TaintBochs. TaintBochs tracks sensitive data by "tainting" it at the hardware level. Tainting information is then propagated across operating system, language, and application boundaries, permitting analysis of sensitive data handling at a whole system level.

We have used TaintBochs to analyze sensitive data handling in several large, real world applications. Among these were Mozilla, Apache, and Perl, which are used to process millions of passwords, credit card numbers, etc. on a daily basis. Our investigation reveals that these applications and the components they rely upon take virtually no measures to limit the lifetime of sensitive data they handle, leaving passwords and other sensitive data scattered throughout user and kernel memory. We show how a few simple and practical changes can greatly reduce sensitive data lifetime in these applications.

1 Introduction

Examining sensitive data lifetime can lend valuable insight into the security of software systems. When studying data lifetime we are concerned with two primary issues: how long a software component (e.g. operating system, library, application) keeps data it is processing alive (i.e. in an accessible form in memory or persistent storage) and where components propagate data (e.g. buffers, log files, other components).

As data lifetime increases so does the likelihood of exposure to an attacker. Exposure can occur by way of an attacker gaining access to system memory or to persistent storage (e.g. swap space) to which data has leaked. Careless data handling also increases the risk of data exposure via interaction with features such as logging, command histories, session management, crash dumps or crash reporting [6], interactive error reporting, etc.

Unfortunately, even simple questions about data lifetime can be surprisingly difficult to answer in real systems. The same data is often handled by many different components, including device drivers, operating system, system libraries, programming language runtimes, applications, etc., in the course of a single transaction. This limits the applicability of traditional static and dynamic program analysis techniques, as they are typically limited in scope to a single program, often require program source code, and generally cannot deal with more than one implementation language.

To overcome these limitations we have developed a tool based on whole-system simulation called TaintBochs, which allows us to track the propagation of sensitive data at hardware level, enabling us to examine *all* places that sensitive data can reside. We examine systems with TaintBochs by running the entire software stack, including operating system, application code, etc. inside a simulated environment. Every byte of system memory, device state, and relevant processor state is tagged with a taint-status flag. Data is "tainted" if it is considered sensitive.

TaintBochs propagates taint flags whenever their corresponding values in hardware are involved in an operation. Thus, tainted data is tracked throughout the system as it flows through kernel device drivers, user-level GUI widgets, application buffers, etc. Tainting is introduced when sensitive data enters the system, such as when a password is read from the keyboard device, an application reads a particular data set, etc.

We applied TaintBochs to analyzing the lifetime of password information in a variety of large, real-world applications, including Mozilla, Apache, Perl, and Emacs on the Linux platform. Our analysis revealed that these applications, the kernel, and the libraries that they relied upon generally took no steps to reduce data lifetime. Buffers containing sensitive data were deallocated without being cleared of their contents, leaving sensitive data to sit on the heap indefinitely. Sensitive data was left in cleartext in memory for indeterminate periods without good reason, and unnecessary replication caused excessive copies of password material to be scattered all over the heap. In the case of Emacs our analysis also uncovered an interaction between the keyboard history

mechanism and shell mode which caused passwords to be placed into the keyboard history in the clear.

On a positive note our analysis revealed that simple modifications could yield significant improvements. For example, adding a small amount of additional code to clear buffers in the string class destructor in Mozilla greatly reduced the amount of sensitive input form data (e.g. CGI password data) in the heap without a noticeable impact on either code complexity or performance.

Our exposition proceeds as follows. In section 2 we present the motivation for our work, discussing why data lifetime is important to security, why minimizing data lifetime is challenging, and how whole system simulation can help. Section 3 describes the design of TaintBochs, its policy for propagating taint information and the rationale behind it, its support for introducing and logging taints, and our analysis framework. Section 4 describes our experiments on Mozilla, Apache, Perl, and Emacs, analyzes the results, and describes a few simple changes we made to greatly reduced the quantity of long-lived tainted data in programs we examined. Section 5 covers related work. Section 6 describes our thoughts about future work in this area. Finally, section 7 concludes.

2 Motivation

This section examines why data lifetime is important, how this issue has been overlooked in many of today's systems, why it is so difficult to ensure minimal data lifetime, and how TaintBochs can help ameliorate these problems.

Threat Model or Why Worry about Data Lifetime? The longer sensitive data resides in memory, the greater the risk of exposure. A long running process can easily accumulate a great deal of sensitive data in its heap simply by failing to take appropriate steps to clear that memory before `free()`ing it. A skillful attacker observing such a weakness could easily recover this information from a compromised system simply by combing an application's heap. More importantly, the longer data remains in memory the greater its chances of being leaked to disk by swapping, hibernation, a virtual machine being suspended, a core dump, etc.

Basic measures for limiting the lifetime of sensitive data including password and key material and keeping it off persistent storage have become a standard part of secure software engineering texts [29] and related literature [13, 28]. Extensive work has been done to gauge the difficulty of purging data from magnetic media once it has been leaked there [11], and even issues of persistence in solid state storage have been examined [12]. Concern about sensitive data being leaked to disk has fueled work on encrypted swap [21] and encrypted file systems [4] which can greatly reduce the impact of sensitive data leaks to disk. Unfortunately, these measures have seen fairly limited deployment.

Identifying long-lived data is not so obviously useful as, say, detecting remotely exploitable buffer overflows. It is a more subtle issue of ensuring that principles of conservative design have been followed to minimize the impact of a compromise and decrease the risk of harmful feature interactions. The principles that underly our motivation are: first, minimize available privilege (i.e. sensitive data access) throughout the lifetime of a program; second, defense in depth, e.g. avoid relying solely on measures such as encrypted swap to keep sensitive data off disk.

While awareness of data lifetime issues runs high among the designers and implementers of cryptographic software, awareness is low outside of this community. This should be a significant point for concern. As our work with Mozilla in particular demonstrates, even programs that should know better are entirely careless with sensitive data. Perhaps one explanation for this phenomenon is that if data is not explicitly identified as, for example, a cryptographic key, it receives no special handling. Given that most software has been designed this way, and that this software is being used for a wide range of sensitive applications, it is important to have an easy means of identifying which data is sensitive, and in need of special handling.

Minimizing Data Lifetime is Hard The many factors which affect data lifetime make building secure systems a daunting task. Even systems which strive to handle data carefully are often foiled by a variety of factors including programmer error and weaknesses in components they rely upon. This difficulty underscores the need for tools to aid examining systems for errors.

Common measures taken to protect sensitive data include zeroing out memory containing key material as soon as that data is no longer needed (e.g. through the C `memset()` function) and storing sensitive material on pages which have been pinned in memory (e.g. via the UNIX `mmap()` or `mlock()` system calls), to keep them off of persistent storage. These measures can and have failed in a variety of ways, from poor interactions between system components with differing assumptions about data lifetime to simple programmer error.

A very recent example is provided by Howard [14] who noted that `memset()` alone is ineffective for clearing out memory with any level of optimization turned on in Borland, Microsoft, and GNU compilers. The problem is that buffers which are being `memset()` to clear their contents are effectively "dead" already, i.e. they will never be read again, thus the compiler marks this

code as redundant and removes it. When this problem was revealed it was found that a great deal of software, including a variety of cryptographic libraries written by experienced programmers, had failed to take adequate measures to address this. Now that this problem has been identified, multiple ad-hoc ways to work around this problem have been developed; however, none of them is entirely straightforward or foolproof.

Sometimes explicitly clearing memory is not even possible. If a program unexpectedly halts without clearing out sensitive data, operating systems make no guarantees about when memory will be cleared, other than it will happen before the memory is allocated again. Thus, sensitive data can live in memory for a great deal of time before it is purged. Similarly, socket buffers, IPC buffers, and keyboard input buffers, are all outside of programmer control.

Memory locking can fail for a wide range of reasons. Some are as simple as memory locking functions that provide misleading functionality. For example, a pair of poorly documented memory locking functions in some versions of Windows, named `VirtualLock()` and `VirtualUnlock()`, are simply advisory, but this has been a point of notable confusion [13].

OS hibernation features do not respect memory locking guarantees. If programs have anticipated the need, they can usually request notification before the system hibernates; however, most programs do not.

Virtual machine monitors such as VMware Workstation and ESX [30] have limited knowledge of the memory management policies of their guest OSes. Many VMM features, including virtual memory (i.e. paging), suspending to disk, migration, etc., can write any and all state of a guest operating system to persistent storage in a manner completely transparent to the guest OS and its applications. This undermines any efforts by the guest to keep memory off of storage such as locking pages in memory or encrypting the swap file.

In addition to these system level complications, unexpected interactions between features within or across applications can expose sensitive data. Features such as logging, command histories, session management, crash dumps/crash reporting, interactive error reporting, etc. can easily expose sensitive data to compromise.

Systems are made of many components that application designers did not develop and whose internals they have little a priori knowledge of. Further, poor handling of sensitive data is pervasive. While a few specialized security applications and libraries are quite conservative about their data handling, most applications, language runtimes, libraries and operating system are not. As we discuss later in Section 4, even the most common components such as Mozilla, Apache, Perl, and Emacs and even the Linux kernel are relatively profligate with their handling of sensitive data. This makes building systems which are conservative about sensitive data handling extremely difficult.

Whole System Simulation can Help TaintBoch's approach of tracking sensitive data of interest via whole system simulation is an attractive platform for tackling this problem. It is practical, relatively simple to implement (given a simulator), and possesses several unique properties that make it particularly well suited to examining data lifetime.

TaintBochs's whole system view allows interactions between components to be analyzed, and the location of sensitive data to be easily identified. Short of this approach, this is a surprisingly difficult problem to solve. Simply `grep`ing for a sensitive string to see if it is present in system memory will yield limited useful information. In the course of traversing different programs, data will be transformed through a variety of encodings and application specific data formats that make naive identification largely impossible. For example, in section 4 we find that a password passing from keyboard to screen is alternately represented as keyboard scan codes, plain ASCII, and X11 scan codes. It is buffered as a set of single-character strings, and elements in a variety of circular queues.

Because TaintBochs tracks data at an architectural level, it does not require source code for the components that an analysis traverses (although this does aid interpretation). Because analysis is done at an architectural level, it makes no assumptions about the correctness of implementations of higher level semantics. Thus, high level bugs or misfeatures (such as a compiler optimizing away `memset()`) are not overlooked.

Comparison of a whole system simulation approach with other techniques is discussed further in the related work, section 5.

3 TaintBochs Design and Implementation

TaintBochs is our tool for measuring data lifetime. At its heart is a hardware simulator that runs the entire software stack being analyzed. This software stack is referred to as the *guest system*. TaintBochs is based on the open-source IA-32 simulator Bochs v2.0.2 [5]. Bochs itself is a full featured hardware emulator that can emulate a variety of different CPUs (386, 486, or Pentium) and I/O devices (IDE disks, Ethernet card, video card, sound card, etc.) and can run unmodified *x*86 operating systems including Linux and Windows.

Bochs is a *simulator*, meaning that guest code never runs directly on the underlying processor—it is merely interpreted, turning guest hardware instructions into appropriate actions in the simulation software. This per-

mits incredible control, allowing us to augment the architecture with taint propagation, extend the instruction set, etc.

We have augmented Bochs with three capabilities to produce TaintBochs. First, we provide the ability to track the propagation of sensitive data through the system at a hardware level, i.e. tainting. Second, we have added logging capabilities that allow system state such as memory and registers at any given time during a system's execution history to be examined. Finally, we developed an analysis framework that allows information about OS internals, debug information for the software that is running, etc. to be utilized in an integrated fashion to allow easy interpretation of tainting information. This allows us to trace tainted data to an exact program variable in an application (or the kernel) in the guest, and code propagating tainting to an exact source file and line number.

Our basic usage model consists of two phases. First, we run a simulation in which sensitive data (e.g. coming from the keyboard, network, etc.) is identified as tainted. The workload consists of normal user interaction, e.g. logging into a website via a browser. In the second phase, the simulation data is analyzed with the analysis framework. This allows us to answer open-ended queries about the simulation, such as where tainted data came from, where it was stored, how it was propagated, etc.

We will begin by looking at the implementation of TaintBochs, focusing on modifications to the simulator to facilitate tainting, logging, etc. We will then move on to examine the analysis framework and how it can be used with other tools to gain a complete picture of data lifetime in a system.

3.1 Hardware Level Tainting

There are two central issues to implementing hardware level tainting: first, tracking the location of sensitive state in the system, and, second, deciding how to evolve that state over time to keep a consistent picture of which state is sensitive. We will examine each of these issues in turn.

Shadow Memory To track the location of sensitive data in TaintBochs, we added another memory, set of registers, etc. called a *shadow memory*. The shadow memory tracks taint status of every byte in the system. Every operation performed on machine state by the processor or devices causes a parallel operation to be performed in shadow memory, e.g. copying a word from register A to location B causes the state in the shadow register A to be copied to shadow location B. Thus to determine if a byte is tainted we need only look in the corresponding location in shadow memory.

If any bit in a byte is tainted, the entire byte is considered tainted. Maintaining taint status at a byte granularity is a conservative approximation, i.e. we do not ever lose track of sensitive data, although some data may be unnecessarily tainted. Bit granularity would take minimal additional effort, but we have not yet encountered a situation where this would noticeably aid our analysis.

For simplicity, TaintBochs only maintains shadow memory for the guest's main memory and the IA-32's eight general-purpose registers. Debug registers, control registers, SIMD (e.g. MMX, SSE) registers, and flags are disregarded, as is chip set and I/O device state. Adding the necessary tracking for other processor or I/O device state (e.g. disk, frame buffer) would be quite straightforward, but the current implementation is sufficient for many kinds of useful analysis. We are not terribly concerned about the guest's ability to launder taint bits through the processor's debug registers, for example, as our assumption is that software under analysis is not intentionally malicious.

Propagation Policy We must decide how operations in the system should affect shadow state. If two registers A and B are added, and one of them is tainted, is the register where the result are stored also tainted? We refer to the collective set of policies that decide this as the *propagation policy*.

In the trivial case where data is simply copied, we perform the same operation in the address space of shadow memory. So, if the assignment $A \leftarrow B$ executes on normal memory, then $A \leftarrow B$ is also executed on shadow memory. Consequently, if B was tainted then A is now also tainted, and if B was not tainted, A is now also no longer tainted.

The answer is less straightforward when an instruction produces a new value based on a set of inputs. In such cases, our simulator must decide on whether and how to taint the instruction's output(s). Our choices must balance the desire to preserve any possibly interesting taints against the need to minimize spurious reports, i.e. avoid tainting too much data or uninteresting data. This roughly corresponds to the false negatives vs. false positives trade-offs made in other taint analysis tools. As we will see, it is in general impossible to achieve the latter goal perfectly, so some compromises must be made.

Processor instructions typically produce outputs that are some function of their inputs. Our basic propagation policy is simply that *if any byte of any input value is tainted, then all bytes of the output are tainted.* This policy is clearly *conservative* and errs on the side of tainting too much. Interestingly though, with the exception of cases noted below, we have not yet encountered any obviously spurious output resulting from our policy.

Propagation Problems There are a number of quite common situations where the basic propagation policy presented before either fails to taint interesting information, or taints more than strictly necessary. We have discovered the following so far:

- *Lookup Tables*. Sometimes tainted values are used by instructions as indexes into non-tainted memory (i.e. as an index into a lookup table). Since the tainted value *itself* is not used in the final computation, only the lookup value it points to, the propagation policy presented earlier would not classify the output as tainted.

  This situation arises routinely. For example, Linux routinely remaps keyboard device data through a lookup table before sending keystrokes to user programs. Thus, user programs never directly see the data read in from the keyboard device, only the non-tainted values they index in the kernel's key remapping table.

  Clearly this is not what we want, so we augmented our propagation policy to handle tainted indexes (i.e. tainted pointers) with the following rule: *if any byte of any input value that is involved in the address computation of a source memory operand is tainted, then the output is tainted, regardless of the taint status of the memory operand that is referenced*.
- *Constant Functions*. Tainted values are sometimes used in computations that always produce the same result. We call such computations *constant functions*. An example of such a computation might be the familiar IA-32 idiom for clearing out a register: `xor eax, eax`. After execution of this instruction, `eax` always holds value `0`, regardless of its original value.

  For our purposes, the output of constant functions never pose a security risk, even with tainted inputs, since the input values are not derivable from the output. In the `xor` example above, it is no less the situation as if the programmer had instead written `mov eax, 0`. In the `xor` case, our naive propagation policy taints the output, and in the `mov` case, our propagation policy does not taint the output (since immediate inputs are never considered tainted).

  Clearly, our desire is to never taint the output of constant functions. And while this can clearly be done for special cases like `xor eax, eax` or similar sequences like `sub eax, eax`, this cannot be done in general since the general case (of which the `xor` and `sub` examples are merely degenerate members) is an arbitrary sequence of instructions that ultimately compute a constant function. For example, assuming `eax` is initially tainted, the sequence:

```
mov ebx, eax   ; ebx = eax
add ebx, ebx   ; ebx = 2 * eax
shl eax, 1     ; eax = 2 * eax
xor ebx, eax   ; ebx = 0
```

  Always computes (albeit circuitously) zero for `ebx`, regardless of the original value of `eax`. By the time the instruction simulation reaches the `xor`, it has no knowledge of whether its operands have the same value because of some deterministic computation or through simple chance; it must decide, therefore, to taint its output.

  One might imagine a variety of schemes to address this problem. Our approach takes advantage of the semantics of tainted values. For our research, we are interested in tainted data representing secrets like a user-typed password. Therefore, we simply define by fiat that we are only interested in taints on non-zero values. As a result, any operation that produces a zero output value never taints its output, since zero outputs are, by definition, uninteresting.

  This simple heuristic has the consequence that constant functions producing nonzero values can still be tainted. This has not been a problem in practice since constant functions themselves are fairly rare, except for the degenerate ones that clear out a register. Moreover, tainted inputs find their way into a constant function even more rarely, because tainted memory generally represents a fairly small fraction of the guest's overall memory.
- *One-way Functions*. Constant functions are an interesting special case of a more general class of computations we call *one-way functions*. A one-way function is characterized by the fact that its input is not easily derived from its output. The problem with one-way functions is that tainted input values generally produce tainted outputs, just as they did for constant functions. But since the output value gives no practical information about the computation's inputs, it is generally uninteresting to flag such data as tainted from the viewpoint of analyzing information leaks, since no practical security risk exists.

  A concrete example of this situation occurs in Linux, where keyboard input is used as a source of entropy for the kernel's random pool. Data collected into the random pool is passed through various mixing functions, which include cryptographic hashes like SHA-1. Although derivatives of the original keyboard input are used by the kernel when it extracts entropy from the pool, no practical information can be gleaned about the original keyboard input from looking at the random number outputs (at least, not *easily*).

  Our system does not currently try to remove tainted outputs resulting from one-way functions, since instances of such taints are few and easily iden-

tifiable. Moreover, such taints are often useful for identifying the spread of tainted data, for example, the hash of a password is often used as a cryptographic key.

Evading Tainting While the propagation policy defined above works well for us in practice, data can be propagated in a manner that evades tainting. For example, the following C code,

```
if (x == 0) y = 0;
else if (x == 1) y = 1;
...
else if (x == 255) y = 255;
```

effectively copies `x` to `y`, but since TaintBochs does not taint comparison flags or the output of instructions that follow a control flow decision based on them, the associated taint for `x` does not propagate to `y`. Interestingly, the Windows 2000 kernel illustrates this problem when translating keyboard scancodes into unicode.

Another possible attack comes from the fact that TaintBochs never considers instruction immediates to be tainted. A guest could take advantage of this by dynamically generating code with proper immediate values that constructs a copy of a string.

Because such attacks do exist, TaintBochs can never prove the absence of errors; we don't expect to use it against actively malicious guests. Instead, TaintBochs is primarily focused on being a testing and analysis tool for finding errors.

Taint Sources TaintBochs supports a variety of methods for introducing taints:

- *Devices*. I/O devices present an excellent opportunity to inject taints into the guest, since they represent the earliest point of the system at which data can be introduced. This is a crucial point, since we are interested in the way a whole system handles sensitive data, even the kernel and its device drivers. TaintBochs currently supports tainting of data at the keyboard and network devices. Support for other devices is currently under development.[1]

  Keyboard tainting simply taints bytes as they are read from the simulated keyboard controller. We use this feature, for example, to taint a user-typed password inside a web browser (see section 4.1.1 for details). This features is essentially binary: keyboard tainting is either on or off.

  Tainting data at the Ethernet card is a slightly more complicated process. We do not want to simply taint entire Ethernet packets, because Ethernet headers, TCP/IP headers, and most application data are of little interest to us. To address this we provide the network card with one or more patterns before we begin a simulation. TaintBochs scans Ethernet frames for these patterns, and if it finds a match, taints the bytes that match the pattern. These taints are propagated to memory as the frame is read from the card. Although this technique can miss data that should be tainted (e.g. when a string is split between two TCP packets) it has proved sufficient for our needs so far.
- *Application-specific*. Tainting at the I/O device level has as its chief benefit the fact that it undercuts all software in the system, even the kernel. However this approach has limitations. Consider, for example, the situation where one wants to track the lifetime and reach of a user's password as it is sent over the network to an SSH daemon. As part of the SSH exchange, the user's password is encrypted before being sent over the network, thus our normal approach of pattern matching is at best far more labor intensive, and less precise than we would like.

  Our current solution to this situation, and others like it, is to allow the *application* to decide what is interesting or not. Specifically, we added an instruction to our simulated IA-32 environment to allow the guest to taint data: `taint eax`. Using this we can modify the SSH daemon to taint the user's password as soon as it is first processed. By pushing the taint decision-making up to the application level, we can skirt the thorny issue that stopped us before by tainting the password after it has been decrypted by the SSH server. This approach has the unfortunate property of being invasive, in that it requires modification of guest code. It also fails to taint encrypted data in kernel and user buffers, but such data is less interesting because the session key is also needed to recover sensitive data.

3.2 Whole-System Logging

TaintBochs must provide some mechanism for answering the key questions necessary to understand taint propagation: *Who has tainted data? How did they get it?* and *When did that happen?*. It achieves this through *whole-system logging*.

Whole system logging records sufficient data at simulation time to reconstitute a fairly complete image of the state of a guest at any given point in the simulation. This is achieved by recording all changes to interesting system state, e.g. memory and registers, from the system's initial startup state. By combining this information with the initial system image we can "play" the log forward

[1] Support for disk tainting and frame buffer tainting is currently underway. With this addition we hope to more completely understand when data is leaked to disk and its lifetime there. We anticipate this will be complete before publication.

to give us the state of the system at any point in time.

Ideally, we would like to log all changes to state, since we can then recreate a perfect image of the guest at a given instant. However, logging requires storage for the log and has runtime overhead from logging. Thus, operations which are logged are limited to those necessary to meet two requirements. First we need to be able to recreate guest memory and its associated taint status at any instruction boundary to provide a complete picture of what was tainted. Second, we would like to have enough register state available to generate a useful *backtrace* to allow deeper inspection of code which caused tainting.

To provide this information the log includes writes to memory, changes to memory taint state, and changes to the stack pointer register (ESP) and frame pointer register (EBP). Each log entry includes the address (EIP) of the instruction that triggered the log entry, plus the instruction count, which is the number of instructions executed by the virtual CPU since it was initialized.

To assemble a complete picture of system state TaintBochs dumps a full snapshot of system memory to disk each time logging is started or restarted. This ensures that memory contents are fully known at the log's start, allowing subsequent memory states to be reconstructed by combining the log and the initial snapshot.

Logging of this kind is expensive: at its peak, it produces about 500 MB/minute raw log data on our 2.4 GHz P4 machines, which reduces about 70% when we add `gzip` compression to the logging code. To further reduce log size, we made it possible for the TaintBochs user to disable logging when it is unneeded (e.g. during boot or between tests). Even with these optimizations, logging is still slow and space-consuming. We discuss these overheads further in section 6.

3.3 Analysis Framework

Taint data provided by TaintBochs is available only at the hardware level. To interpret this data in terms of higher level semantics, e.g. at a C code level, hardware level state must be considered in conjunction with additional information about software running on the machine. This task is performed by the analysis framework.

The analysis framework provides us with three major capabilities. First, it answers the question of which data is tainted by giving the file name and line number where a tainted variable is defined. Second, it provides a list of locations and times identifying the code (by file name and line number) which caused a taint to propagate. By browsing through this list the causal chain of operations that resulted in taint propagation can be unraveled. This can be walked through in a text editor in a fashion similar to a list of compiler errors. Finally, it provides the ability to inspect any program that was running in the guest at any point in time in the simulation using `gdb`. This allows us to answer any questions about tainting that we may not have been able to glean by reading the source code.

Traveling In Time The first capability our analysis framework integrates is the ability to scroll back and forth to any time in the programs execution history. This allows the causal relationship between different tainted memory regions to be established, i.e. it allows us to watch taints propagate from one region of memory to the next. This capability is critical as the sources of taints become untainted over time, preventing one from understanding what path data has taken through the system simply by looking at a single point.

We have currently implemented this capability through a tool called `replay` which can generate a complete and accurate image of a simulated machine at any instruction boundary. It does this by starting from a snapshot and replaying the memory log. It also outputs the physical addresses of all tainted memory bytes and provides the values of EBP and ESP, exactly, and EIP, as of the last logged operation. EBP and ESP make backtraces possible and EIP is identifies the line of code that caused tainting (e.g. copied tainted data). `replay` is a useful primitive, but it still presents us with only raw machine state. To determine what program or what part of the kernel owns tainted data or what code caused it to be tainted we rely on another tool called `x-taints`.

Identifying Data A second capability of the analysis framework is matching raw taint data with source-level entities in user code, currently implemented through a tool called `x-taints`, our primary tool for interpreting tainting information. It combines information from a variety of sources to produce a file name and line number where a tainted variable was defined.

`x-taints` identifies static kernel data by referring to `System.map`, a file produced during kernel compilation that lists each kernel symbol and its address. Microsoft distributes similar symbol sets for Windows, and we are working towards integrating their use into our analysis tools as well.

`x-taints` identifies kernel heap allocated data using a patch we created for Linux guests that appends source file and line number information to each region allocated by the kernel dynamic memory allocator `kmalloc()`. To implement this we added extra bytes to the end of every allocated region to store this data. When run against a patched kernel, this allows `x-taints` to display such information in its analysis reports.

`x-taints` identifies data in user space in several steps. First, `x-taints` generates a table that maps

physical addresses to virtual addresses for each process. We do this using a custom extension to Mission Critical's `crash`, software for creating and analyzing Linux kernel crash dumps. This table allows us to identify the process or processes that own the tainted data. Once `x-taints` establishes which process owns the data it is interested in, `x-taints` turns to a second custom `crash` extension to obtain more information. This extension extracts a core file for the process from the physical memory image on disk. `x-taints` applies `gdb` to the program's binary and the core file and obtains the name of the tainted variable.

For analysis of user-level programs to be effective, the user must have previously copied the program's binary, with debugging symbols, out of the simulated machine into a location known to `x-taints`. For best results the simulated machine's libraries and their debugging symbols should also be available.

Studying Code Propagating Taints The final capability that the analysis framework provides is the ability to identify which code propagated taints, e.g. if a call to `memcpy` copies tainted data, then its caller, along with a full backtrace, can be identified by their source file names and line numbers.

`x-taints` discovers this by replaying a memory log and tracking, for every byte of physical memory, the PID of the program that last modified it, the virtual address of the instruction that last modified it (EIP), and the instruction count at which it was modified.[2] Using this data, `x-taints` consults either `System.map` or a generated core file and reports the function, source file, and line number of the tainting code.

`x-taints` can also bring up `gdb` to allow investigation of the state of any program in the simulation at any instruction boundary. Most of the debugger's features can be used, including full backtraces, inspecting local and global variables, and so on. If the process was running at the time of the core dump, then register variables in the top stack frame will be inaccurate because only EBP and ESP are recorded in the log file. For processes that are not running, the entire register set is accurately extracted from where it is saved in the kernel stack.

4 Exploring Data Lifetime with TaintBochs

Our objective in developing TaintBochs was to provide a platform to explore the data lifetime problem in depth in real systems. With our experimental platform in place, our next task was to examine the scope of the data lifetime in common applications.

In applying TaintBochs we concerned ourselves with three primary issues:

- *Scope*. Where was sensitive data was being copied to in memory.
- *Duration*. How long did that data persist?
- *Implications*. Beyond the mere presence of problems, we wanted to discover how easy they would be to solve, and what the implications were for implementing systems to minimize data lifetime.

There is no simple answer to any of these questions in the systems we analyzed. Data was propagated all over the software stack, potential lifetimes varied widely, and while a wide range of data lifetime problems could be solved with small changes to program structure, there was no single silver bullet. The one constant that did hold was that careful handling of sensitive data was almost universally absent.

We performed three experiments in total, all of them examining the handling of password data in a different contexts. Our first experiment examined Mozilla [27], a popular open source web browser. Our second experiment tests Apache [1], by some reports the most popular server in the world, running a simple CGI application written in Perl. We believe these first two experiments are of particular interest as these platforms process millions of sensitive transactions on a daily basis. Finally, our third experiment examines GNU Emacs [26], the well-known text-editor-turned-operating-system, used by many as their primary means of interaction with UNIX systems.

In section 4.1 we describe the design of each of our experiments and report where in the software stack we found tainted data. In section 4.2 we analyze our results in more detail, explaining the lifetime implications of each location where sensitive data resided (e.g. I/O buffers, string buffers). In section 4.3 we report the results of experiments in modifying the software we previously examined to reduce data lifetime.

4.1 Experimental Results

4.1.1 Mozilla

In our first experiment we tracked a user-input password in Mozilla during the login phase of the Yahoo Mail website.

Mozilla was a particularly interesting subject not only because of its real world impact, but also because its size. Mozilla is a massive application ($\sim$3.7 million lines of code) written by many different people, it also has a huge number of dependencies on other components (e.g. GUI toolkits).

[2] An earlier version recorded the physical address corresponding to EIP, instead of PID plus virtual address. This unnecessarily complicated identifying the process responsible when a shared library function (e.g. `memmove`) tainted memory.

Given its complexity, Mozilla provided an excellent test of TaintBoch's ability to make a large application comprehensible. TaintBochs passed with flying colors. One of us was able to analyze Mozilla in roughly a day. We consider this quite acceptable given the size of the data set being analyzed, and that none of us had prior familiarity with its code base.

For our experiment, we began by a booting a Linux[3] guest inside TaintBochs. We then logged in as an unprivileged user, and started X with the `twm` window manager. Inside X, we started Mozilla and brought up the webpage `mail.yahoo.com`, where we entered a user name and password in the login form. Before entering the password, we turned on TaintBoch's keyboard tainting, and afterward we turned it back off. We then closed Mozilla, logged out, and closed TaintBochs.

When we analyzed the tainted regions after Mozilla was closed, we found that many parts of the system fail to respect the lifetime sensitivity of the password data they handle. The tainted regions included the following:

- *Kernel random number generator*. The Linux kernel has a subsystem that generates cryptographically secure random numbers, by gathering and mixing entropy from a number of sources, including the keyboard. It stores keyboard input temporarily in a circular queue for later batch processing. It also uses a global variable `last_scancode` to keep track of the previous key press; the keyboard driver also has a similar variable `prev_scancode`.
- *XFree86 event queue*. The X server stores user-input events, including keystrokes, in a circular queue for later dispatch to X clients.
- *Kernel socket buffers*. In our experiment, X relays keystrokes to Mozilla and its other clients over Unix domain sockets using the `writev` system call. Each call causes the kernel to allocate a `sk_buff` socket structure to hold the data.
- *Mozilla strings*. Mozilla, written in C++, uses a number of related string classes to process user data. It makes no attempt to curb the lifetime of sensitive data.
- *Kernel tty buffers*. When the user types keyboard characters, they go into a `struct tty_struct` "flip buffer" directly from interrupt context. (A flip buffer is divided into halves, one used only for reading and the other used only for writing. When data that has been written must be read, the halves are "flipped" around.) The key codes are then copied into a tty, which X reads.

[3]We conducted our experiment on a Gentoo [10] Linux guest with a 2.4.23 kernel. The guest used XFree86 v4.3.0r3 and Mozilla v1.5-r1.

4.1.2 Apache and Perl

In our second experiment, we ran Apache inside TaintBochs, setting it up to grant access to a CGI script written in Perl. We tracked the lifetime of a password entered via a simple form and passed to a trivial CGI script.

Our CGI script initialized Perl's CGI module and output a form with fields for user name, password, and a submit button that posted to the same form. Perl's CGI module automatically parses the field data passed to it by the browser, but the script ignores it. This CGI script represents the minimum amount of tainting produced by Perl's CGI module as any CGI script that read and used the password would almost certainly create extra copies of it.

In this experiment, the web client, running outside TaintBochs, connected to the Apache server running inside. TaintBochs examined each Ethernet frame as it entered the guest, and tainted any instance of a hard-coded password found in the frame. This technique would not have found the password had it been encoded, split between frames, or encrypted, but it sufficed for our simple experiment.

Using Apache version 1.3.29 and Perl version 5.8.2, we tracked the following sequence of taints as we submitted the login form and discovered that the taints listed below persist after the request was fully handled by Apache and the CGI program:

- *Kernel packet buffers*. In function `ne_block_input`, the Linux kernel reads the Ethernet frame from the virtual NE2000 network device into a buffer dynamically allocated with `kmalloc`. The frame is attached to an `sk_buff` structure used for network packets. As we found with Unix domain sockets in the Mozilla experiment, the kernel does not zero these bytes when they are freed, and it is difficult to predict how soon they will be reused.
- *Apache input buffers*. When Apache reads the HTTP request in the `ap_bread` function, the kernel copies it from its packet buffer into a buffer dynamically allocated by Apache. The data is then copied to a stack variable by the CGI module in function `cgi_handler`. Because it is on the stack, the latter buffer is reused for each CGI request made to a given Apache process, so it is likely to be erased quickly except on very low-volume web servers.
- *Apache output buffer*. Apache copies the request to a dynamically allocated output buffer before sending it to the CGI child process.
- *Kernel pipe buffer*. Apache flushes its output buffer to the Perl CGI subprocess through a pipe, so tainted data is copied into a kernel pipe buffer.
- *Perl file input buffer*. Perl reads from the pipe into a

dynamically allocated file buffer, 4 kB in size. The buffer is associated with the file handle and will not be erased as long as the file is open and no additional I/O is done. Because Apache typically sends much less than 4 kB of data through the pipe, the read buffer persists at least as long as the CGI process.

- *Perl string buffers*. Perl copies data from the input buffer into a Perl string, also dynamically allocated. Furthermore, in the process of parsing, the tainted bytes are copied into a second Perl string.

All of these buffers contain the full password in cleartext.

4.1.3 Emacs

In our third experiment we tracked the lifetime of a password entered into `su` by way of Emacs's shell mode.

At its core GNU Emacs is a text editor. Because it is built on top of a specialized Lisp interpreter, modern versions can do much more than edit text. Indeed, many users prefer to do most of their work within Emacs.

Many of the functions Emacs performs may involved handling sensitive data, for example, activities that might prompt for passwords include interacting with shells, browsing web pages, reading and sending email and newsgroup articles, editing remote files via `ssh`, and assorted cryptographic functionality.

We chose Emacs' "shell mode" for our first investigation. In shell mode, an Emacs buffer becomes an interface to a Unix shell, such as `bash`, running as an Emacs subprocess. Emacs displays shell output in the buffer and passes user input in the buffer to the shell. Emacs does not implement most terminal commands in the shell buffer, including commands for disabling local echo, so passwords typed in response to prompts by `ssh`, `su`, etc. would normally echo. As a workaround, shell mode includes a specialized facility that recognizes password prompts and reads them without echo in a separate "minibuffer." We decided to investigate how thoroughly Emacs cleared these passwords from its memory after passing them to the subprocess.

To start the experiment, we booted a guest running the Debian GNU/Linux "unstable" distribution, logged in as an unprivileged user, and started Emacs. Within Emacs, we started shell mode and entered the `su` command at the shell prompt.[4] Using the TaintBochs interface, we enabled tainting of keyboard input, typed the root password, and then disabled keyboard input tainting. Finally, we closed the shell sessions, exited Emacs, logged off, and shut down TaintBochs.

Using the generated memory and taint logs, we ran a taint analysis at a point soon after the `su` subshell's prompt had appeared in the Emacs buffer. The results identified several tainted regions in Emacs and the kernel:

- *Kernel random number generator and keyboard data*. See the Mozilla experiment (section 4.1.1) for more information.
- *Global variable* `kbd_buffer`. All Emacs input passes through this buffer, arranged as a circular queue. Each buffer element is only erased after approximately 4,096 further input "events" (keyboard or mouse activities) have occurred.
- *Data referenced by global variable* `recent_keys`. This variable keeps track of the user's last 100 keystrokes.
- *Each character in the password, as a 1-character Lisp string*. Lisp function `comint-read-noecho` accumulates the password string by converting each character to a 1-character string, then concatenating those strings. These strings are unreferenced and will eventually be recycled by the garbage collector, although when they will be erased is unpredictable (see appendix A for further discussion).
- *The entire password as a Lisp string*. The password is not cleared after it is sent to the subprocess. This string is also unreferenced.
- *Stack*. Emacs implements Lisp function calls in terms of C function calls, so the password remains on the process stack until it is overwritten by a later function call that uses as much stack.
- *Three kernel buffers*. When the user types keyboard characters, they go into a `struct tty_struct` "flip buffer" directly from interrupt context. The key codes are then copied into a tty that Emacs reads, and then into a second tty when Emacs passes the password to its shell subprocess.

The password typed can be recovered from any of these tainted regions. The tainted strings are of particular interest: the Emacs garbage collector, as a side effect of collecting unreferenced strings, will erase the first 4 bytes (8 bytes, on 64-bit architectures) of a string. Thus, several of the taints above would have shrunk or disappeared entirely had we continued to use Emacs long enough for the garbage collector to be invoked.

Finally, as part of our investigation, we discovered that entering a special Emacs command (`view-lossage`) soon after typing the password would actually reveal it on-screen in plaintext form. This behavior is actually documented in the Emacs developer documentation for `comint-read-noecho`, which simply notes that "some people find this worrysome [sic]." Because this piece of advice is not in the Emacs manual, a typical Emacs user would never see it. The same developer documentation also says that,

[4] Given the superuser's password, `su` opens a subshell with superuser privileges.

"Once the caller uses the password, it can erase the password by doing `(fillarray STRING 0)`," which is untrue, as we can see from the above list of taints.

4.1.4 Windows 2000 Workloads

To illustrate the generality of data lifetime problems, our fourth experiment consisted of two workloads running on Windows 2000.

We first examined the process of logging into a Windows 2000 machine. By tainting keyboard input while typing the user's password at Windows' initial login dialog, we found at least two occurrences of the password in memory after the login process was completed: a tainted scancode representation and a unicode representation.

Our second workload mirrors the web login experiment we ran with Mozilla on Linux (see section 4.1.1). In this workload, we used Internet Explorer 5.0 under Windows 2000. We again found a tainted scancode representation of the password sitting in memory after the login process was complete.

We have forgone further analysis as a lack of application and OS source code limited our ability to diagnose the cause of taints and discern how easily they could be remedied.

4.2 Analysis of Results

This section discusses the results found in the previous sections and discusses the data lifetime implications of each major class of tainting result found. For a more in-depth discussion of the data lifetime implications of different storage classes (e.g. stack, heap, dynamically allocated vs. garbage collected), the reader should see appendix A.

4.2.1 Circular Queues

Circular queues of events are common in software. Circular queue data structures are usually long-lived and often even statically allocated. Data in a circular queue survives only as long as it takes the queue to wrap around, although that may be a long time in a large or inactive queue.

Our experiments uncovered three queues that handle tainted data: the Linux kernel random number generator batch processing queue (described in more detail in section 4.2.4 below), XFree86's event queue, and Emacs' event queue.

In each case we encountered, tainted data was stored in plaintext form while it awaited processing. More importantly, in each case, after inputs were consumed, they were simply left on the queue until they were eventually overwritten when the queue head wrapped around. Because each queue processes keyboard input, these factors present a non-deterministic window of opportunity for an attacker to discover keys typed, since keystrokes are left in the queue even after they have been consumed.

We can significantly reduce data lifetime in each of the cases encountered simply by zeroing input after it has been consumed. In section 4.3, we describe application of such a fix to Emacs.

4.2.2 I/O Buffers

Buffers are more transient and thus tend to be allocated on the heap or, occasionally, the stack. Buffers are sometimes created for use in only a single context, as with the case of kernel network buffers. In other cases, they survive as long as an associated object, as in the case of kernel pipe buffers and some Apache input buffers.

Our experiments encountered many kinds of tainted input and output buffer data. In the Mozilla experiment, we found tainted tty buffers and Unix domain socket buffers; in the Apache and Perl experiment, we found tainted kernel network buffers, Apache input and output buffers, kernel pipe buffers, and Perl file input buffers.

There is no simple bound on the amount of time before freed buffer data will be reallocated and erased. Even if an allocator always prefers to reuse the most recently freed block for new allocations ("LIFO"), some patterns of allocate and free operations, such as a few extra free operations in a sequence that tends to keep the same amount of memory allocated, can cause sensitive data to linger for excessive amounts of time. Doug Lea's `malloc()` implementation, used in `glibc` 2.*x* and elsewhere, actually has far more complex behavior that actually tends toward "FIFO" behavior in some circumstances (see Appendix A for more details). Heap fragmentation can also extend sensitive data lifetime.

We can solve the problem of sensitive data in I/O buffers by zeroing them when they are no longer needed. Because relatively large I/O buffers of 4 kB or more are often allocated even for a few bytes, only space in the buffer that was actually filled with data should be zeroed.

4.2.3 Strings

Tainted strings appeared in the results of all three of our experiments: in Mozilla, C++ string classes; in Perl, Perl strings; in Emacs, Lisp strings.

String data tends to be allocated on the heap or, occasionally, the stack. Strings are often used in operations that copy data, such as concatenation or substring operations. This can lead their contents to be replicated widely in the heap and the stack.

This type of replication was especially prevalent in the cases we encountered because of the high-level nature of the string representations used. In each case, the

```
NS_IMETHODIMP
nsTextControlFrame::CheckFireOnChange()
{
  nsString value;
  GetText(&value);
  //different fire onchange
  if (!mFocusedValue.Equals(value))
  {
    mFocusedValue = value;
    FireOnChange();
  }
  return NS_OK;
}
```

Figure 1: In this example Mozilla needlessly replicates sensitive string data in the heap. nsString's constructor allocates heap space and `GetText(&value)` taints that data. This extra copy is unnecessary merely to do a comparison.

programmer need not be aware of memory allocation and copying. Indeed, Perl and Emacs Lisp provide no obvious way to determine that string data has been reallocated and copied. Normally this is a convenience, but for managing the lifetime of sensitive data it is a hazard.

We discovered that this problem is especially vexing in Mozilla, because there are many easy pitfalls that can end up making heap copies of strings. Figure 1 illustrates this situation with a snippet of code from Mozilla that ends up making a heap copy of a string just to do a string comparison (`nsString` is a string class that allocates storage from the heap). This needlessly puts another copy of the string on the heap and could have been accomplished through a variety of other means as fundamentally string comparison does not require any additional allocation.

Because, like buffer data, tainted strings tend to occupy heap or stack space, the considerations discussed in the previous section for determining how long freed data will take to be cleared also apply to string data. In practice the pattern of lifetimes is likely to differ, because buffers are typically fixed in size whereas strings vary widely.

4.2.4 Linux Random Number Generator

In both the Mozilla and Emacs experiments we discovered tainted data in the Linux kernel associated with its cryptographically secure random number generator (RNG). The source of this tainting was keyboard input which is used as a source of randomness. The locations tainted fell into three categories.

First, the RNG keeps track of the user's last keystroke in static variable `last_scancode` so that repeated keystrokes from holding down a key are not used as a source of randomness. This variable holds only one keystroke and is overwritten on subsequent key press, thus it is a source of limited concern.

Second, to avoid doing expensive hash calculations in interrupt context, the RNG stores plaintext keystrokes into a 256-entry circular queue `batch_entropy_pool` and processes them later in a batch. The same queue is used for batching other sources of randomness, so the length of the window of opportunity to recover data from this queue depends heavily on workload, data lifetime could vary from seconds to minutes on a reasonably loaded system to hours or even days on a system left suspended or hibernated.

Third, the RNG's entropy pools are tainted. These are of little concern, because data is added to the pools only via "mixing functions" that would be difficult or impossible for an attacker to invert.

4.3 Treating the Taints

4.3.1 Mozilla

Mozilla makes no attempt to reduce lifetime of sensitive form data, however, simple remedies exist which can help significantly.

First, uses of `nsString` for local variables (as in Figure 1) can be replaced with variables of type `nsAutoString`, a string class that derives buffer space from the same storage class as the string itself, thus, data in stack based storage will not be propagated to the heap. This practice is actually recommended by Mozilla coding guidelines, so the example code snippet in Figure 1 ought to have incorporated this change.

One often legitimately needs to have a heap-allocated string e.g. in string members of a dynamically allocated object. Therefore, to reduce data lifetime in this case classes should zero out their contents when they are destroyed. This trivial change to the string class's destructor significantly reduces the lifetime of sensitive data, without inducing any perceptible change in program performance.

To evaluate the impact of this approach we added zeroing to string destructors in Mozilla, and reran our experiments. We found this small change was very successful in reducing both the amount of tainted data and its lifetime. With this patch, the amount of tainted data in Mozilla's address space reduced in half, and taints from destroyed string objects were completely eliminated.

Figure 2 illustrates this point by showing the amount of tainted string data in Mozilla's address space as a function of time (as measured in tens of millions of instructions elapsed since the start of tainting). The spike in both runs marks when the user has submitted the web form containing their password. During this time, Mozilla does considerable processing on the password:

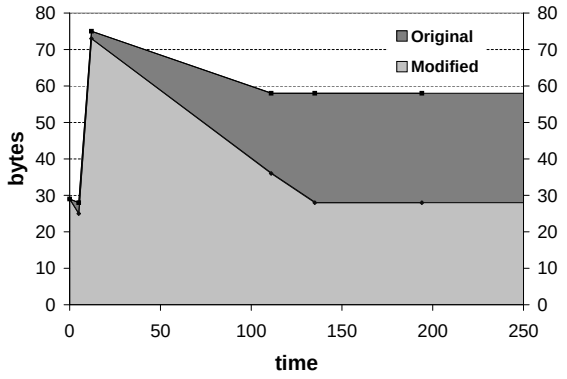

Figure 2: A comparison of the amount of tainted string data in the original Mozilla versus our modified version. Our zero-on-free string remedy reduces tainted string data by half in the steady state.

it is touched by GUI widgets, HTML form handling code, and even the browser's JavaScript engine.

String data is progressively deallocated by Mozilla as it finishes the form submission process and begins loading the next page. As Figure 2 shows, the amount of tainted data is reduced by roughly half once Mozilla hits a steady state. The difference between the original and modified runs is entirely accounted for by garbage heap data from Mozilla's various string classes.

The baseline of tainted data bytes in the modified run is accounted for by explicit `char*` copies made from string classes. This means that our patch entirely eliminated tainted data resulting from destroyed string objects in our experiment, and highlighted the places where Mozilla made dangerous explicit `char*` string copies.

4.3.2 Emacs

As with Mozilla, we modified Emacs to reduce the number of long-lived tainted regions. We made two changes to its C source code, each of which inserted only a single call to `memset`. First, we modified `clear_event`, a function called to clear input events as they are removed from the input queue. The existing code only set events' type codes to `no_event`, so we added a line to zero the remainder of the data.

Second, we modified `sweep_strings`, called by the garbage collector to collect unreferenced strings. The existing code zeroed the first 4 bytes (8 bytes, on 64-bit architectures) of strings as a side effect. We modified it to zero all bytes of unreferenced strings.

We reran the experiment with these modifications, forcing garbage collection after entering the password. This had the desired effect: all of the tainted, unreferenced Lisp strings were erased, as were all of the tainted input buffer elements. We concluded that relatively simple changes to Emacs can have a significant impact on the lifetime of sensitive data entrusted to it.

5 Related Work

Previous work on whole system simulation for analyzing software has largely focused on studying performance and providing a test bed for new hardware features. Extensive work on the design of whole system simulators including performance, extensibility, interpretation of hardware level data in terms of higher level semantics, etc. was explored in SimOS [22].

Dynamic binary translators which operate at the single process level instead of the whole system level have demonstrated significant power for doing dynamic analysis of software [8]. These systems work as assembly-to-assembly translators, dynamically instrumenting binaries as they are executed, rather than as complete simulators. For example, Valgrind [19] has been widely deployed in the Linux community and provides a wide range of functionality including memory error detection (à la Purify [15]), data race detection, cache profiling, etc. Somewhere between an full simulator and binary translator is Hobbes [7], a single process *x*86 interpreter that can detect memory errors and perform runtime type checking. Hobbes and Valgrind both provide frameworks for writing new dynamic analysis tools.

Dynamo [3] is an extremely fast binary translator, akin to an optimizing JIT compiler intended to be run during program deployment. It has been used to perform dynamic checks to enhance security at runtime by detecting deviations from normal execution patterns derived via static analysis. This technique has been called program shepherding [16]. It is particularly interesting in that it combines static analysis with dynamic checking.

These systems have a narrower scope than TaintBochs as they operate on a single program level, but they offer significant performance advantages. That said, binary translators that can operate at the whole system level with very high efficiency have been demonstrated in research [31] and commercial [18] settings. The techniques demonstrated in TaintBochs could certainly be applied in these settings.

The term "tainting" has traditionally referred to tagging data to denote that the data comes from an untrusted source. Potential vulnerabilities are then discovered by determining whether tainted data ever reaches a sensitive sink. This of course differs from our use of taint information, but the fundamental mechanism is the same. A tainted tag may be literally be a bit associated with data,

as in systems like TaintBochs or Perl's tainting or may simply be an intuitive metaphor for understanding the results of a static analysis.

Perl [20] provides the most well known example of tainting. In Perl, if "tainting" is enabled, data read by built-in functions from potentially untrusted sources, i.e. network sockets, environment variables, etc. is tagged as tainted. Regular expression matching clears taint bits and is taken to mean that the programmer is has checked that the input is "safe." Sensitive built-in functions (e.g. `exec`) will generate a runtime error if they receive tainted arguments.

Static taint analysis has been applied by a variety of groups with significant success. Shankar et al. [24] used their static analysis tool Percent-S to detect format string vulnerabilities based on a tainting style analysis using type qualifier inference and programmer annotations. Scrash [6], infers which data in a system is sensitive based on programmer annotations to facilitate special handling of that data to allow secure crash dumps, i.e. crash dumps which can be shipped to the application developer without revealing users sensitive data. This work is probably the most similar to ours in spirit as its focus is on making a feature with significant impact on sensitive data lifetime safe. The heart of both of these systems is the CQual [23], a powerful system for supporting user extensible type inference.

Ashcraft et al. [2] successfully applied a tainting style static analysis in the context of their meta-compilation system with extremely notable success. In the context of this work they were able to discover a large number of new bugs in the Linux and OpenBSD kernels. Their system works on a more ad-hoc basis, effectively and efficiently combining programmer written compiler extensions with statistical techniques.

Static analysis and whole system simulation both have significant strengths and can be used in a complementary fashion. Both also present a variety of practical trade-offs. Static analysis can examine all paths in a program. As it need not execute every path in the program to glean information about its properties, this allows it to avoid an exponential "blow up" in possible execution paths. This can be achieved through a variety of means, most commonly by making the analysis insensitive to control flow. On the other hand, simulation is basically program testing with a very good view of the action. As such, it examines only execution paths that are exercised.

Static analysis is typically performed at the source code level, thus all code is required to perform the analysis, and the analysis typically cannot span multiple programs. Further, most but not all static analysis tools require some program annotation to function. Whole system simulation can be easily used to perform analysis of properties that span the entire software stack and can be essentially language independent. Possession of source code is not even required for an analysis to include a component, although it is helpful for interpreting results.

One clear advantage of dynamic analysis in general is that it actually allows the program to be run to determine its properties. Because many program properties are formally undecidable they cannot be discovered via static analysis alone. Also, because lower level analysis works at the architectural level, it makes no assumptions about the correctness of implementations of higher level semantics. Thus, higher level bugs or misfeatures (such as a compiler optimizing away `memset()` as described in section 2) are not overlooked.

6 Future Work

Many questions remain to be answered about data lifetime. There is no current empirical work on how long data persists in different memory region types (e.g. stack, heap, etc.) under different workloads. As discussed in Appendix A allocation policies are quite complicated and vary widely, making it difficult to deduce their impact from first principles. This problem also holds for virtual memory subsystems. While our framework identifies potential weaknesses well, we would like a more complete solution for gaining quantitative information about data lifetime in the long term (over hours, and even days) under different workloads both in memory and on persistent storage.

One direction for similar inquiries might be to examine data lifetime with a more accurate simulation, such as one that would reflect the physical characteristics of the underlying devices à la work by Gutmann [11, 12].

Another area for future work is improving our simulation platform. Speed is a fundamental limitation of TaintBochs' current incarnation because of the fine-grained tainting and detailed logging that it does. TaintBochs can run as much as 2 to 10 times slower than Bochs itself. The enormity of the logging done by TaintBochs also presents a problem for our postmortem analysis tools, since it can easily take minutes or hours to replay a memory log to an interesting point in time.

We have several ideas for optimizing our system. By reducing the volume of data we log, or simply doing away with our dependency on logging altogether, we could vastly improve TaintBochs overheads. The whole-system logging technique used in ReVirt [9], for example, only had a 0-8% performance cost.

Reduced logging overhead also opens up the possibility of moving TaintBochs functionality onto faster whole-system simulation environments like those discussed in section 5. The right trade-offs could allow us to do TaintBochs-like analysis in production scenarios.

7 Conclusion

Minimizing data lifetime greatly decreases the chances of sensitive data exposure. The need for minimizing the lifetime of sensitive data is supported by a significant body of literature and experience, as is the recognition of how difficult it can be in practice.

We explored how whole system simulation can provide a practical solution to the problem of understanding data lifetime in very large and complex software systems through the use of hardware level taint analysis.

We demonstrated the effectiveness of this solution by implementing a whole system simulation environment called TaintBochs and applying it to analyze sensitive data lifetime in a variety of large real world applications.

We used TaintBochs to study sensitive data lifetime in real world systems by examining password handing in Mozilla, Apache, Perl, and Emacs. We found that these systems and the components that they rely on handle data carelessly, resulting in sensitive data being propagated widely across memory with no provisions made to purge it. This is especially disturbing given the huge volume of sensitive data handled by these applications on a daily basis. We further demonstrated that a few practical changes could drastically reduce the amount of long lived sensitive data in these systems.

8 Acknowledgments

This work was supported in part by the National Science Foundation under Grant No. 0121481 and a Stanford Graduate Fellowship.

References

[1] Apache Software Foundation. The Apache HTTP Server project. `http://httpd.apache.org`.

[2] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *IEEE Symposium on Security and Privacy*, May 2002.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

[4] M. Blaze. A cryptographic file system for UNIX. In *ACM Conference on Computer and Communications Security*, pages 9–16, 1993.

[5] Bochs: The cross platform IA-32 emulator. `http://bochs.sourceforge.net/`.

[6] P. Broadwell, M. Harren, and N. Sastry. Scrash: A system for generating secure crash information. In *Proceedings of the 11th USENIX Security Symposium*, August 2003.

[7] M. Burrows, S. N. Freund, and J. Wiener. Run-time type checking for binary programs. *International Conference on Compiler Construction*, April 2003.

[8] B. Cmelik and D. Keppel. Shade: a fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 128–137. ACM Press, 1994.

[9] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Operating Systems Review*, 36(SI):211–224, 2002.

[10] Gentoo Linux. `http://www.gentoo.org`.

[11] P. Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the 6th USENIX Security Symposium*, july 1996.

[12] P. Gutmann. Data remanence in semiconductor devices. In *Proceedings of the 7th USENIX Security Symposium*, Jan. 1998.

[13] P. Gutmann. Software generation of practically strong random numbers. In *Proceedings of the 8th USENIX Security Symposium*, August 1999.

[14] M. Howard. Some bad news and some good news. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode%/html/secure10102002.asp`, October 2002.

[15] IBM Rational software. IBM Rational Purify. `http://www.rational.com`.

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

[17] D. Lea. A memory allocator. `http://gee.cs.oswego.edu/dl/html/malloc.html`.

[18] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, February 2002.

[19] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In O. Sokolsky and M. Viswanathan, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.

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

[21] N. Provos. Encrypting virtual memory. In *Proceedings of the 10th USENIX Security Symposium*, pages 35–44, August 2000.

[22] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: The SimOS approach. *IEEE Parallel and Distributed Technology: Systems and Applications*, 3(4):34–43, Winter 1995.

[23] J. S. *Type Qualifiers: Lightweight Specifications to Improve Software Quality*. PhD thesis, University of California, Berkeley, December 2002.

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

[25] D. A. Solomon and M. Russinovich. *Inside Microsoft Windows 2000*. Microsoft Press, 2000.

[26] R. Stallman et al. GNU Emacs. `ftp://ftp.gnu.org/pub/gnu/emacs`.

[27] The Mozilla Organization. Home of the mozilla, firebird, and camino web browsers. `http://www.mozilla.org/`.

[28] J. Viega. Protecting sensitive data in memory. `http://www-106.ibm.com/developerworks/security/library/s-data.html?dwzo%ne=security`.

[29] J. Viega and G. McGraw. *Building Secure Software*. Addison-Wesley, 2002.

[30] VMware, Inc. VMware virtual machine technology. `http://www.vmware.com/`.

[31] E. Witchel and M. Rosenblum. Embra: Fast and flexible machine simulation. In *Measurement and Modeling of Computer Systems*, pages 68–79, 1996.

A Data Lifetime by Memory Region Type

Most data in software can be classified in terms of its allocation discipline as static, dynamic, or stack data. Allocation and release of each kind of data occurs in a different way: static data is allocated at compile and link time, dynamic data is allocated explicitly at runtime, and stack data is allocated and released at runtime according to an implicit stack discipline. Similarly, taints in each kind of data are likely to persist for different lengths of time according to its allocation class. The allocators used in various operating systems vary greatly, so the details will vary from one system to another. To show the complexity of determining when freed memory is likely to be reallocated, we describe the reallocation behavior of Linux and the GNU C library typically used on it:

- *Static data.* Static data persists at least as long as the process itself. How much longer depends on the operating system and the system's activity level. The Linux kernel in particular takes a very "lazy" approach to clearing pages. As with most kernels, pages are not zeroed when they are freed, but unlike some others (such as Windows NT [25] and descendants) pages are not zeroed in a background thread either. Pages are not zeroed when memory is requested by a process, either. Only when a process first tries to access an allocated page will Linux actually allocate and zero a physical page for its use. Therefore, under Linux static data persists after a process's termination as long as it takes the kernel to reassign its page to another process. (Pages reclaimed from user process may also be allocated by the kernel for its own use, but in that case they may not be zeroed immediately or even upon first write.)

  When allocation and zeroing does become necessary, the Linux kernel's "buddy allocator" for pages is biased toward returning recently freed pages. However, its actual behavior is difficult to predict, because it depends on the system's memory allocation pattern. When single free pages are coalesced into larger free blocks by the buddy allocator, they are less likely to be returned by new allocation requests for single pages. They are correspondingly more likely to be returned for multi-page allocations of the proper size, but those are far rarer than single-page allocations.
- *Dynamic data.* Dynamic data only needs to persist until it is freed, but it often survives significantly longer. Few dynamic memory allocators clear memory when it is freed; neither the Linux kernel dynamic memory allocator (`kmalloc()`) nor the `glibc` 2.x dynamic memory allocator (`malloc()`) zeroes freed (or reallocated) memory. The question then becomes how soon the memory is reassigned on a new allocation. This is of course system-dependent. In the case of Linux, the answer differs between the kernel and user-level memory allocators, so we treat those separately.

  The Linux kernel "slab" memory allocator draws each allocation from one of several "pools" of fixed-size blocks. Some commonly allocated types, such as file structures, have their own dedicated pools; memory for other types is drawn from generic pools chosen based on the allocation size. Within each pool, memory is allocated in LIFO order, that is, the most recently freed block is always the first one to be reused for the next allocation.

  The GNU C library, version 2.x, uses Doug Lea's implementation of `malloc()` [17], which also pools blocks based on size. However, its behavior is far more complex. When small blocks (less than 512 bytes each) are freed, they will be reused if allocations of identical size are requested immediately. However, any allocation of a large block (512 bytes or larger) causes freed small blocks to be coalesced into larger blocks where possible. Otherwise, allocation happens largely on a "best fit" basis. Ties are broken on a FIFO basis, that is, *less* recently freed blocks are preferred. In short, it is difficult to predict when any given free block will be reused. Dynamic data that is never freed behaves in a manner essentially equivalent to static data.
- *Stack data.* Data on a process's stack changes constantly as functions are called and return. As a result, an actively executing program should tend to clear out data in its stack fairly quickly. There are some important exceptions. Many programs have some kind of "main loop" below which they descend rarely, often only to terminate execution. Data on the stack below that point tends to remain for long periods. Second, some programs occasionally allocate large amounts of stack space e.g. for input or output buffers (see 4.1.2). Such data may only be fully cleared out by later calls to the same routine, because other routines are unlikely to grow the stack to the point that much of the buffer is cleared. If data read into large buffers on the stack is sensitive, then it may be long-lived. Data that remains on the stack at program termination behaves the same way as static data.

Most of the accounts above only describe when memory tends to reallocated, not when it is cleared. These are not the same because in most cases, reallocated memory is not necessarily cleared by its new owner. Memory used as an input or output buffer or as a circular queue may only be cleared as it is used and perhaps not at all (by this owner) if it is larger than necessary. Padding bytes in C structures, inserted by the programmer manually or the compiler automatically, may not be cleared either.