Taint Analysis in Practice

Automated Counter-measures on the World Wide Web

Michiel Scholten
mbscholt@few.vu.nl

Vrije Universiteit Amsterdam

Bachelor Computer Science

December 11, 2007
Abstract

In this paper I will look into various software and hardware based attack-detection and repair mechanisms. These mechanisms try to protect software systems against exploits by, for example, detecting a control-flow modification and stopping the process. The emphasis will lay on techniques using taint analysis and the various aspects of this kind of systems. Generation of signature for filtering malicious network data and automatic patch generation based on signatures of these attacks are the next steps to protect the software against future attacks.
Contents

1 Introduction ................................................. 3
   1.1 Overview of the kinds of threats/attacks .................. 3
      1.1.1 Attack vectors .................................. 3
   1.2 Host-based intrusion detection ........................... 4
      1.2.1 Dynamic taint analysis ................................................. 5
      1.2.2 Alternatives .................................................. 5
   1.3 Network based mechanisms ................................. 6
   1.4 Why taint analysis ........................................... 6
   1.5 Signatures ..................................................... 6
   1.6 Patches ....................................................... 7
   1.7 Overview ..................................................... 7

2 Taint analysis ................................................ 8
   2.1 Tainting data ................................................. 8
      2.1.1 Granularity .................................................. 9
      2.1.2 When to untaint certain data ......................... 9
   2.2 Process vs. full-system analysis .......................... 10

3 Implementations ............................................. 11
   3.1 Per-process monitoring .................................... 11
      3.1.1 TaintCheck ................................................. 11
      3.1.2 Vigilante .................................................. 12
   3.2 System-wide monitoring .................................... 12
      3.2.1 Argos ..................................................... 12
      3.2.2 Minos, a hardware-based implementation .............. 13
   3.3 Encrypted channels ......................................... 14

4 Limitations and Optimisations ................................. 15
   4.1 Speed ......................................................... 15
      4.1.1 Sweeper ................................................... 16
      4.1.2 Eudaemon .................................................. 16
      4.1.3 Alex Ho et al. ............................................. 17
      4.1.4 STEM/Application Communities ......................... 18
   4.2 What taint analysis does not do ............................ 18
      4.2.1 What is not taken into account ......................... 18
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 Signatures</td>
<td>20</td>
</tr>
<tr>
<td>5.1 Vigilante</td>
<td></td>
</tr>
<tr>
<td>5.1.1 Dynamic dataflow analysis</td>
<td>20</td>
</tr>
<tr>
<td>5.1.2 Self-Certifying Alerts</td>
<td>21</td>
</tr>
<tr>
<td>5.2 Argos</td>
<td></td>
</tr>
<tr>
<td>5.2.1 SweetBait</td>
<td>22</td>
</tr>
<tr>
<td>5.3 Prospector</td>
<td></td>
</tr>
<tr>
<td>6 Conclusion</td>
<td>26</td>
</tr>
<tr>
<td>Bibliography</td>
<td>27</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Anti-virus software and authors of software themselves are always a step behind the software threatening online computers. Be it your average Windows PC or a server running all kinds of services, there’s software – or people – out there trying to exploit the programs running on it. The costs of the failure of the computers we rely on can go up to six million dollars per hour [34]. A virus scanner is default on Windows machines nowadays, but it depends on signatures for known threats and generally some additional heuristics; new exploits and data manipulation can be overseen, and the protection always lags behind. Also, clever manipulation of data by a hacker goes unnoticed by such software. Systems that keep track of the processes’ control flow and the data those programs use for their operations can detect a possible attack (like when data from the network is used as jump targets, function addresses or instructions). By analysing the program code and the data used by the attack, a signature of the exploit can be generated, with which in turn patches can be automatically generated to prevent a future exploit.

In this thesis, we will explore host-based intrusion detection of new attacks by means of dynamic taint analysis (DTA). We will describe how DTA works, what its advantages and disadvantages are, and how how it can be used for automatic generation of signatures and patches.

1.1 Overview of the kinds of threats/attacks

There is of course a wide range of possible attacks targeting software (services). It is good to signify the difference between a vulnerability and an exploit. A vulnerability is a type of bug that can be used by an attacker to alter the intended operation of the software in a malicious way. An exploit is an actual input that triggers a software vulnerability, typically with malicious intent and devastating consequences [3]. Virus scanners for example use signatures of exploits, whereas it is more effective to use signatures of the vulnerability being exploited.

1.1.1 Attack vectors

Various ways of intrusions exist. They can come in the form of code-injection attacks, control-flow changes or they can be designed to steal data. Buffer overflow attacks became notorious after the Morris Worm in 1988 [37].
Stack smashing attacks (stack overflows) are attacks where bugs allowing an overflow of a stack buffer to overwrite a function's return address are exploited. Because the attacker can make the return address point to some other place, arbitrary code can be executed. Examples of projects trying to protect software from stack smashing attacks are StackGuard [7], Stackshield and gcc extensions [12, 7].

Heap corruption attacks (heap overflows) can be more powerful than stack overflows, allowing an attacker to overwrite an arbitrary memory location, and as a result execute arbitrary code. This is done, for example, by providing a bigger chunk of data then initially advertised so the receiving program does not reserve a large enough buffer. This way, a block containing a memory location or function pointer can be overwritten by that chunk. A more advanced form has the attacker overflowing link pointers that are used to maintain a structure keeping free regions. This way, an attacker can overwrite virtually any memory location with any data [1]. The cause of this is memory allocation functions being implemented to store control data together with the actual allocated memory.

Format string attacks involve a feature in the printf() family of functions. This feature allows the number of characters printed to be stored in a location in memory. An attacker can manipulate the string provided to overwrite any location in memory with arbitrary values. These attacks are more flexible to use, as they offer more options to the attacker, including overwriting function arguments, such as the file to be executed with the execve() system call.

Double-free attacks can be performed when the program frees a pointer that was already freed. “Double-free errors do not share the characteristics of heap-corruption attacks in the sense that they do not overflow a buffer, and so when considering the analysis they require special treatment” [36].

Overwriting or moving around function (instruction) pointers and overwriting control structures – like heap values of pointers – with a value provided by the attacker can of course lead to the process executing wholly different functions than normal execution flow would have.

Besides being able to inject code or change the execution flow, an attacker can also cause a denial of service in the software by letting it go into an undefined state and have it freeze, for example. Sometimes this is the only attack a vulnerability allows, but it is still undesired behaviour.

Launching an attack is made easy by black-hat hacker tools and security tools like the Metasploit framework[1] and the Milw0rm[2] library. For on-going news on security issues, Bugtraq[3] is a nice starting point.

1.2 Host-based intrusion detection

There are two areas where malicious activity can be detected: in the network and on the host. Here, we will describe some mechanisms for host-based intrusion detection. Host-based detection

---

2. Milw0rm, [http://www.milw0rm.com/](http://www.milw0rm.com/)
3. Bugtraq mailing list, [http://www.securityfocus.com/archive/1](http://www.securityfocus.com/archive/1)
mechanisms tend to be more accurate because of more information about the processes involved is available, and the mechanism has a lot more control over the system. This thesis will be about dynamic taint analysis, but alternatives are present.

1.2.1 Dynamic taint analysis

Dynamic taint analysis (DTA) is a mechanism which tracks incoming data from the network throughout the process. Data originating from the network is untrusted, and therefore marked ‘tainted’. Operations on this data are kept an eye on, and taint flags are propagated to the result of such operations. When data from a tainted piece of memory is used in an important operation, for example as target address in a jump, an alert is raised and relevant action is performed. DTA has the potential to be very accurate, but this comes at the cost of performance because a lot of emulation is involved.

1.2.2 Alternatives

A relatively cheap way of preventing control-flow attacks is using address space randomisation, instruction set randomisation [20] and randomising parts of the operating system, such as the location of the stack [5], the location of the heap [5], the system call interface [5]. This is already being done in modern operating systems, like Linux Address Space Randomisation is enabled by default since kernel 2.6.12, OpenBSD and Windows Vista. As it has a relatively low overhead, this method can be used on production servers. When it is combined with the replaying of a recorded attack on such a server, taint analysis can be offloaded to a dedicated server, so the production server has no performance penalty [40].

Examples of instruction set randomisation include the mentioned [20]. They create process-specific randomised instruction sets, applying Kerckhoff’s principle. This principle states that a cryptosystem should be secure even if everything about the system – except the key – is public knowledge. You need to know the key to the randomisation algorithm, so if an attacker injects code, it will likely be invalid for the randomised processor, so it causes a runtime exception. The project modified the Linux kernel, the GNU binutils tools and, the bochs-x86 emulator; this all leads to a significant performance penalty, but implementing in hardware, like the Transmeta Crusoe (of which the microcode can be modified to implement a different instruction set) should deliver a usable system.

StackGuard

StackGuard provides compiler extensions (patches to gcc) that let a program enter a fail-safe state in case of misuse of a buffer. This enables a low-overhead protection scheme for existing code, making the software safe from buffer overflow attacks (especially stack smashing). StackGuard prevents changes to the return address while a function is still active. It does so by either completely preventing writes to the return address, or detecting changes to the address before function return. The latter is more efficient and portable, while the former is more secure. StackGuard can adaptively switch between both modes.

A ‘canary’ word is placed next to the return address on the stack. The value is checked right before the function returns. Preventing the change of the return address is done with the

\footnote{Address Space Randomisation is enabled by default since kernel 2.6.12}

\footnote{A direct descendent of the Welsh miner’s canary}
MemGuard tool [8]. This tool can protect certain pieces of memory, like the return address while the function is running. For more information, see the StackGuard article [7].

Others

Keeping a system up-to-date by ‘patching’ it regularly, updating virus scanners and using firewalls may provide a certain protection, but are no alternatives for the systems discussed in this thesis, as it will always be lagging behind the facts.

1.3 Network based mechanisms

Another way of detecting (potential) attacks is by analysing network traffic. If, for example, a sharp rise of port scans on a certain range is detected, further investigation into services listening to these ports might be a good idea. Or it could be that a certain IP range is attacked, targeting a single company or service. Tools like Snort [32] can be used to filter network traffic based on certain packet signatures. However, encryption and pacing techniques may make network analysis really hard.

In addition, some approaches have proposed to simply execute all bytes in the network stream, using an emulator. If the traffic contains a malware, it will start pointing to itself after a given time, trying to perform operations. For an example, see [28].

Network based solutions are outside of the scope of this paper, as it will focus on host-based systems and dynamic taint analysis in particular.

1.4 Why taint analysis

Taint analysis provides various interesting and thorough techniques to detect, stop and counter attacks on vulnerabilities. These vulnerabilities need not to be known in advance: because the system keeps track of the control flow and detects an exploit, it can protect against zero-day attacks. Also, depending of the amount of book keeping performed by the analysis system, there is a whole lot of information available about the vulnerability and the specific exploit.

A lot of intrusions are being done by manipulating input data leading to a process executing unexpected code, or with ‘weird’ data. In fact, more than one third of all vulnerabilities notes reported by US-CERT in 2006 consisted of buffer overflows [11]. An attacker can try to inject code by performing a buffer overrun attack, which lets the program counter walk into that new code, or overwrite a return pointer on the stack so the process will instead continue with a wholly different piece of code (which may be injected in another buffer). Operating systems try to address this issue by building overflow protection into their core, but do not always succeed; Windows Vista for example has shown to be still vulnerable to such attacks [33].

1.5 Signatures

To counter attacks on software and services, various ways of detection and filtering can be used. A common way of filtering at the network level is using Snort [32] rules, which filter packages according to certain patterns. Bro [27] and Cisco’s NBAR system [39] are comparable intrusion
Taint Analysis in Practice

detection systems. Other techniques are Autograph [18], Earlybird [35] and Honeycomb. However, this paper will not go deeply into the subject of these. The emphasis will be on generating signatures using taint analysis.

We need automatic signature generation techniques because manual signature generation is slow and error prone. It is important to be fast, as vulnerabilities can be exploited a lot faster than a human can respond to, like with a worm outbreak.

1.6 Patches

Not only signatures can be generated using analysis of attacks. Patches are a very viable product too. Using taint analysis a system can automatically generate patches (both source and binary) to try to eliminate the vulnerability (for example a buffer overflow). These can be tested using a replay with recorded data, after which they can be shared with other machines.

1.7 Overview

The remainder of this paper is organised as follows. The various techniques behind Taint analysis systems will be discussed in Section 2, various systems will be evaluated in Section 3. Limitations of Taint analysis, speed problems and matter that falls outside of its scope will be discussed in Section 4. In Section 5 the generation and distribution of signatures and patches is briefly explained. This paper will not go deeply into this matter, but will focus on the various aspects of Taint analysis in general.
Chapter 2

Taint analysis

The most common type of remote attacks on the Internet are control data attacks. Internet worms are a good example of this. Taint analysis is a technique designed to detect these attacks, analyse the cause and ultimately generate signatures to help other machines detect the exploit, and generate patches to fix the vulnerability in the first place. The first research to this was already done in 1976 by Dorothy E. Denning [10].

2.1 Tainting data

Taint analysis works by marking certain pieces of memory tainted when it is coming from untrusted sources, like the network. This can be done by adding an integrity bit to every 32-bit word of memory, for example, like Minos does [9]. It then can use Biba’s low-watermark integrity policy [2] with values “high” and “low” to describe the level of threat the data poses. For more information on Biba’s LWM integrity policy, see Section 3.2.2.

It is important to use a balanced policy describing what data should be marked ‘tainted’. Generally, taint analysis systems taint data originating from the network as being tainted, but this data can also be written to disk and read back. Minos keeps track of all tainted data and uses changes in the file system to save taintedness information on disk too [9].

Taint propagation occurs when arithmetics is performed with tainted values, like for example a value $x$ is increased with the tainted variable $n$, the resulting $x$ is also tainted. Same goes for or-ing values etc. When a tainted memory value is loaded into a register, that register is also marked tainted. Bookkeeping of the ‘tainted’ marks is generally done by adding some memory structures containing the flags for its memory section. As an optimisation, paging techniques can be used to keep the overhead low. Some systems also write the taint flags with data written to disk to not loose the tainting information. Systems like TaintCheck can go further, and record the system call number, a snapshot of the then current stack, and a copy of the data that was written, which enables detailed forensics.

Alerts are raised when tainted data is used in the program counter, as value for jumps or in system registers with function similar to the program counter. These are the moments that the untrusted values can become a threat to the execution flow. Overwriting a return address, for example, will not ring bells, but the moment the function tries to use the value to jump, a ‘taint alert’ kicks in. In Figure 2.1 the legit return address of the function is overwritten by data coming from a tainted source, resulting in the value evil_target taking its place, which
Taint Analysis in Practice

will likely be pointing to code the attacker wants to get executed.

![Figure 2.1: A function’s return address gets overwritten by tainted data](image)

2.1.1 Granularity

When tainting data, it is important to make trade-offs about the granularity of the tracking. For every $n$ bytes of memory that’s flagged, a system uses $m$ bits or bytes of data to keep tabs on it. This can be a single bit denoting ‘tainted’ or not, or it can include a pointer to a complete data structure containing the history of what changed during the flow through the process, like what TaintCheck does [26] (see also Section 3.1.1). The system by Alex Ho et al., for example, uses a granularity of one bit per byte. To minimise the memory overhead this causes, all systems have devised a paging scheme to reduce the memory by these bits (or the pointers and data structures like with TaintCheck in extended mode).

Minos uses a ‘tainted’ bit for every 32-bit word [9]. Alex Ho et al.’s system has a taint bit for every byte, where Argos and TaintCheck can store a lot more tainting information, enabling detailed forensics (see for example the TaintCheck Section: 3.1.1).

For performance reasons, certain data types are sometimes ignored by the policy. Examples of these are the EFLAGS register which is frequently affected by operations involving untrusted data (tagging would make it impossible to differentiate between malicious and benevolent sources), floating point data structures and data used with MMX and SSE instructions (related to floating point data-types). Argos, for example can be configured to take MMX instructions in account or not. Empirical evidence is that few (if any) exploits exist that use vulnerabilities in this sort of code [31]. Ignoring them can alleviate the performance penalty a bit, as they do not have to be tracked.

2.1.2 When to untaint certain data

If a register or memory location is initialised to a constant value (zero or another pre-defined value) again, this should be seen as the moment to untaint its value, as the data it contains is not influenced by tainted data anymore. This of course results in fewer false-positives. Untainting data must be done, because otherwise the system would be incorrect and generate false positives. However, as can be seen in Section 5.3 this can lead to gaps when backtracking tainted data. But not untainting data also leads to gaps, only different ones.
2.2 Process vs. full-system analysis

A monitoring system can scrutinise a single process and its data, or could keep tabs on the whole system. The first way is generally less intensive, as it only has to track the data flow in a limited amount of code. However, because of its limited scope, it can miss important clues, like tainted data originating from files that were written by other processes with data that came from the network. TaintCheck for example keeps track of tainted data in files the process has written, but can not know about taintedness of a file it didn’t see being accessed by other processes. A full-system analyser tracks all tainted data, but has the limitation that when an alarm is triggered, it does not know at once which process caused it \[31\]. It takes effort to find out the exact process under attack, but is still feasible.

Also, full-system analysers can handle DMA, memory mapped buffers (e.g., shared buffers with tainted data from another process) and attacks on the kernel and other subsystems. A monitoring system that only tracks a single process misses out on these. To get an idea how Argos works, see Figure \[3.1\].
Chapter 3

Implementations

Various approaches to doing taint analysis exist. The base of the systems use generally the same information flow model [10]. These approaches will be illustrated by various projects and implementations, which each lay the emphasis on another aspect of the process.

3.1 Per-process monitoring

Taint analysis on a per-process base does not protect the OS kernel, but looks at a certain process specifically. This process generally is a service (like an httpd) run on a public server.

3.1.1 TaintCheck

Analysis can be done by keeping track of a single process and monitoring its use of data and execution flow. An example of this is TaintCheck [26], which is based on the open source x86 emulator Valgrind [25]. TaintCheck basically causes the process to run in a sand box. It keeps track of everything the process does, where it gets its data from (e.g., network, certain file paths etc.), where that data is used, with what other data it is combined and where it flows to. It does so by decompiling the binary code and processing the instructions (where Valgrind does the legwork of decompiling). With Valgrind, a program can be instrumented while it is being run through so-called skins. TaintCheck is implemented as such a skin: “whenever program control reaches a new basic block, Valgrind first translates the block of x86 instructions into its own RISC-like instruction set, called UCode. It then passes the UCode block to TaintCheck, which instruments the UCode block to incorporate its taint analysis code. TaintCheck then passes the rewritten UCode block back to Valgrind, which translates the block back to x86 code so that it may be executed. Once a block has been instrumented, it is kept in Valgrind’s cache so that it does not need to be re-instrumented every time it is executed.” [26]

There is also a Windows version of TaintCheck, which is implemented using DynamoRIO [11], another dynamic binary instrumentation tool.

TaintCheck itself exists out of three components: TaintSeed to decide what inputs should be tainted, TaintTracker which takes care of how the taint attribute propagates and TaintAssert, which raises an attack alarm if tainted data is used in a suspicious way. Their default policies can all be configured to the user’s liking. To investigate the optional logs those components create, Exploit Analyzer was designed.
Because it decompiles the binary into RISC-like instructions, it slows server execution between 1.5 and 40 times [26]. This approach however enables TaintCheck to be deployed with any program, without having to modify this program to support taint analysis.

TaintCheck keeps track of tainted data with its TaintSeed component. Every byte of memory, including the registers, stack, heap, etc., has a four-byte shadow memory that stores a pointer to a Taint data structure if that location is tainted, or a NULL pointer if it is not [26]. It uses a page-table-like structure to minimise the overhead of shadow memory, so it uses very little memory in practice. Keeping its policy in mind, TaintSeed examines the arguments and results of each system call, and then decides whether any memory written by the system call should be marked as tainted or untainted. The data structure used when a piece of memory is tainted contains the system call number, a snapshot of the then current stack, and a copy of the data that was written. The Exploit Analyzer can be used to examine this information later on. Also, there is an option to disable logging; the shadow memory then only contains a bit indicating whether the corresponding memory is tainted [26].

3.1.2 Vigilante

Vigilante [6] is a project by Microsoft Research and is similar to TaintCheck. It also monitors single processes and uses virtual addresses. Therefore, it can not handle DMA or memory mappings. An important part of Vigilante is generating and sharing signatures using so-called self-certifying alerts. For more information on this, see Section 5.1.

3.2 System-wide monitoring

To protect the whole system, including the operating system’s kernel, system-wide monitoring can be deployed. With this technique, the whole system runs in a monitored sandbox. This can be a virtualised system, like with Argos [31], or use a hardware-based approach, like with Minos [9].

3.2.1 Argos

Argos [31] works a bit similar to TaintCheck and Vigilante. But where those two systems protect individual processes and leave the kernel and non-monitored services vulnerable, Argos doesn’t monitor a single process but the whole system, using an instrumented version of the x86 emulator Qemu. This x86 emulator “tracks network data throughout execution to identify their invalid use as jump targets, function addresses, instructions, etc. Furthermore, system call policies disallow the use of network data as arguments to certain calls.” [31] It first used only Qemu as basis, but more recent versions use techniques from the paper of Ho et al. in Eurosys 2006 [16] to use both Qemu and Xen to speed things up. See also Section 4.1.3.

Because it runs the whole system in its sandbox and uses an emulator to decompile code, Argos supports any (unmodified) operating system, which enables it to also protect the kernel, device drivers and all processes. This causes slowdown of at least 16 times in comparison with its host. Because it can handle both virtual and physical addresses, Argos can also take into account complex memory operations like memory mapping and DMA (which is quite often ignored by other projects) and handle complex exploits, like register springs [31]. When an attack is detected, it injects its own OS-specific forensics shell-code apart from the exploit code.
to learn more about the exact vulnerability. For a high-level overview, see Figure 3.1. In this figure, network data is entering the system running in the emulator ①, where it is also logged in a trace database. Dynamic taint analysis is used to detect when a vulnerability is exploited ②, which results in an alarm ③, firing the signature generation phase ④-⑥. At step ④, the dumped memory blocks and the additional information obtained by the shellcode inserted by Argos are used for correlation with the network traces in the trace database. This results in a signature ⑤, which is not optimal yet, as it may contain superfluous data. Therefore, Argos submits the signature to the subsystem called SweetBait (see Section 5.2.1) for refinement, with a more generic signature as result.

Furthermore, Argos is able to automatically generate signatures based on the correlation of the exploit’s memory footprint and its network trace ③1. These are automatically distributed.

![Figure 3.1: Argos: High-Level Overview (taken from 31)](image)

### 3.2.2 Minos, a hardware-based implementation

Minos ③9 aims for a hardware-supported solution, describing a framework to implement in the processors and memory system of the computers the secured processes will run on. Its design is intended to speed up the analysis process by profiting from hardware support, which is needed to reduce the performance penalty these systems cause. Minos uses an integrity bit per 32-bits word of data, which results in 33 bits of physical memory used. This also calls for an extra bit added to the data bus, and the integrity bits are also swapped to disk when the relevant memory is paged out, preserving the information. By using Biba’s low-water-mark integrity policy ③2 with values *high* and *low*, the implementation “specifies that any subject may modify any object if the object’s integrity is not greater than that of the subject, but any subject that reads an object has its integrity lowered to the minimum of the object’s integrity and its own.” ③9 This way the system can determine whether the data used is trustworthy or not, and stop execution.

---

③1The Biba Model or Biba Integrity Model is a model, designed such that subjects may not corrupt data in a level ranked higher than the subject, or be corrupted by data from a lower level. In the Low watermark extension, read down is permitted, but the level of the reading object is lowered to the level of the other object.
of a process when low integrity data is used for – for example – a jump or pointers.

Low integrity data can become high integrity because of information flow, as can be seen in the three examples in Figure 3.2.

```java
if (LowIntegrityData == 5)
   HighIntegrityData = 5;

HighIntegrityData =
   HighIntegrityLookupTable[LowIntegrityData];

HighIntegrityData = 0;
while (LowIntegrityData--) {
   HighIntegrityData++;
}
```

Figure 3.2: Propagation of low integrity data to a higher level (taken from [9])

Effective deployment relies on this hardware implementation. A prototype was implemented in the Bochs emulator [9]. However, this implementation is very slow, which is caused for a large part by the inherent slowness of the chosen emulator.

Minos does not generate signatures at all. Because it uses virtual addresses (and can not directly handle physical to virtual address translation at all), it can detect certain exploits, such as a register spring attack [38], but can not directly determine where the attack originated without an inelegant hack.

Other implementations use processor extensions for trusted instruction execution, for example [23].

### 3.3 Encrypted channels

In *Tales from the Crypt* [42], a system being able of dealing with encrypted network communication is proposed. *Hassle* is a honeypot comparable to Prospector, but able to handle encrypted channels. It retaints data on encrypted connections, making the tags point to decrypted data. The honeypot can detect and fingerprint both monomorphic and polymorphic attacks on SSL-encrypted channels, and is application agnostic. Hassle is build on top of Argos [31].

Detecting attacks is made quite hard by the various characteristics of encryption and use of pacing on network links. Looking for certain patterns in network traffic is of no use because of this, so methods like Snort [32] and Bro [27] can not be used.
Chapter 4

Limitations and Optimisations

Of course, all techniques and solutions have shortcomings. Taint analysis is generally suffering from slow execution, and it does not provide a cure-all method for protecting computer programs or complete systems.

4.1 Speed

A serious issue with which most of the projects have to cope, is the speed penalty the monitored software takes. TaintCheck has slowdowns between 1.5 and 40 in comparison to its host [26] (in a non-optimised prototype and largely caused by the Valgrind emulator), the approach of Ho et al. of a factor of 2 up to 100 (which is also caused by the specific application, ssh), Argos a minimum of 15 times (being at most about 2 times as slow as the non-instrumented Qemu it is running on, but only 16% slower in case of Apache and float operations) and of course emulates a complete system, including kernel and other processes. Prospector has an additional overhead of about 16% compared to Argos, with Apache running 18 times slower than the native host. Vigilante does not have clear numbers on the overhead of their detection mechanism (it depends on the kind of worm being detected: 0.7 ms for generating a filter for Slammer, 5.1 ms for Blaster, and 205.3 ms for CodeRed [6]), but has a much small overhead when applying filters, with for example doing a CodeRed attack on IIS causing an overhead of 2.07% by 10 probes per second. Minos aims for a fast implementation by having hardware support, but performs quite badly in their current implementation using Bochs [9], which is for a large part due to the slowness of the per-instruction emulation of that emulator and because it has to emulate the complete system, like Argos. Also, a switch to the faster Qemu is being considered [1]. Application Communities [22] are a way of distributing the emulation load over the various members of the community, which can be brought down to a 6% performance penalty in case of a group of 15,000 members running Apache.

Mind that most of the overhead is being caused by the emulators and decompilers used by the respective implementations.

[http://wwwcsif.cs.ucdavis.edu/~crandall/DIMVANinos.ppt]
4.1.1 Sweeper

Sweeper [40] is a project from the group that also developed the TaintCheck [26] suite. It aims to provide a really low-overhead detection system, and does extended analysis by replaying the snapshotted exploit. It can roll back state of the process multiple times to do the analysis and test the ‘antibody’ it generates. If things are done and the antibody is in place, the process is rolled back one more time to recover from the interception, and the process continues, having the patch against the exploit in place. The whole process of detecting an attack and generating antibodies can be done in under 60 milliseconds, with a 1% overhead during normal execution [40].

The lightweight part of the detection is done by address randomisation. If an exploit is hit, an alarm kicks in and a snapshot is taken. Replaying an attack can be rather difficult though. Especially when time-stamps and encryption are used, it may be quite hard to properly play the steps another time, especially on another machine. Vigilante also suffers from this, as its system is build on replayability.

4.1.2 Eudaemon

Eudaemon aims to blur the borders between protected and unprotected applications [29]. It can attach to any running process and instrument it dynamically by redirecting execution to a user-space emulator. It does so by attaching to the process using the Unix ptrace system call, pausing the execution of the process while saving its processor state, and injecting a small amount of shell-code in its address space. This shell-code is used to call a modified version of the Argos emulator which is linked in as a library, at which point the process is moved into the emulator, which starts with the processor that was previously saved. When desired the process can be moved from being emulated to native execution again, so applications can be moved from and to a protected mode ("honeypot mode") at will. This is important when CPU cycles are in great demand on the machine. Based on need, certain processes can be monitored, and others being run in native mode.

This way of at-will enabling and disabling of monitoring can be used in various ways.

Idle-time honeypots

As PCs tend to be idle for a large part of their uptime [17], part of that time can be used to facilitate a honeypot, just like some screen savers perform calculations on proteins. The honeypot mode can be switched on in idle time to take the existing, running software on the machine and run it in an instrumented fashion, using the machine’s normal IP address(es) [29].

If this is more widely deployed, an attacker can not know in advance whether a certain machine is instrumented at the time he attacks. This way, no pre-compiled list of victims can be used, as a vulnerable host now can use instrumentation on the software later. The attacker risks being exposed so long as a subset of the hosts run a honeypot.

Honey-on-demand

Another way Eudaemon can be used is as ‘the big red button’, i.e., a button that users may press when they are about to access an unknown and possibly suspicious website, or when they
open attachments, view certain files etc. By pressing that button, the application will be run in honeypot mode and be safe against client-side exploits because of the heavy instrumentation.

Applications can also make use of that interface. A mail reader for example could demand to be run in emulation mode when opening a new email message, while running at native speed the rest of the time.

This mode can also be used for servers. For example, when there is no patch yet for a vulnerability, or to overlap the time the administrator needs to test certain patches. When ran in this mode, the service is protected against the exploits. This way of using Eudaemon lets it enter the intrusion prevention area.

Eudaemon is implemented on Linux, but a port to Windows is almost done [29].

4.1.3 Alex Ho et al.

Alex Ho et al. did research to find ways of causing less slowdown of instrumented systems [16]. The result is a system using demand emulation when doing taint-based protection, in which the whole operating system runs virtualised inside the Xen hypervisor. This causes almost no overhead, so no performance hit. However, the whole system is being analysed, and when a region of code deals with tainted data, that part is emulated using the Qemu emulator.

This course-grained combination of virtualisation and full-system emulation allows an essentially new hardware feature to be incorporated in an OS-agnostic manner on commodity systems. The CPU of the virtual machine runs at native speeds. If the processor accesses tainted data, the virtual machine monitor (VMM) switches the virtual machine from the virtual CPU to an emulated processor running as a user-space application in a control VM. This emulator tracks the propagation of tainted data throughout the system. Once the emulated processor ceases to manipulate tainted data, the VMM can revert the system back to virtualised execution [10]. Modications to the computing environment—like tainting data from the network, changes to the behaviour of the CPU, memory and I/O devices—are made outside the protected virtual machine, so no changes to the protected OS or applications are required.

The taint model of this system is similar to the other systems described. Tainted data can flow freely between memory and data registers, but system registers should never be tainted. An attempt of the processor to move tainted data into a system register results in a taint violation, causing a fault, and causes the VM to be switched into emulation mode. The current (virtualised) CPU state is preserved and execution is transferred into a hardware emulator running in user-space in the control VM. The emulator has direct mapped access to the VM’s memory and so continues to execute the machine in place, now emulating instructions that operate on tainted data [10].

The implementation uses a paging scheme to minimise the amount of memory overhead needed for the taint bits (which each describe a single byte). As data is marked as tainted, it is important that all references to it reflect this fact and result in a page fault on access. This approach takes advantage of shadow page tables [13, 14, 15]. This technique involves maintaining a protected version of a VM’s page tables in hypervisor memory, beyond the reach of that VM. These shadow tables are used by the hardware MMU, and so contain authoritative information on a VM’s virtual memory [10].
4.1.4 STEM/Application Communities

Angelos Keromytis’ research group at the University of Columbia has an other approach to counter security risks: they create artificial diversity. The group introduced the concept of Application Communities (AC’s), which are collections of identical programs which collectively monitor for exploits [22]. Every instance instruments a different subset of its code (including overlap) to make a fair and low-overhead monitoring system. Together, the members of an AC work to identify previously unknown flaws or attacks, exchanging the information to build a global protection. Individual members may be attacked successfully, but over a certain time span, the AC as a whole converges to a state of immunity against that exploit or vulnerability. Because it distributes the monitoring task, it has both a broad coverage and a lower overhead per individual system.

The emulator used to instrument specific parts of an member’s code is called the x86-based Selective Transactional EMulator, or STEM. Each node in an AC emulates different “slides” of the application. Detected faults are distributed over the AC, so the other nodes can protect the buggy code area too. Also, already attacked members can be cured with the appropriate fix. STEM can also undo any memory changes inside the function containing the vulnerability and simulate an error return from that function [22].

Because full emulation is so expensive (generally a slowdown of 30 times), the emulation is distributed over the members in the Application Community. With a sufficiently large community (15,000 for Apache for instance), the performance degradation can be as low as 6%, or 73% when only 15 Apache hosts are used. More detailed numbers can be found in Table 4.1 which describes an evaluation of an AC with members running Apache, measuring performance in requests per second for 10000 requests at a concurrency level of 5 for 100 trials [22].

4.2 What taint analysis does not do

Of course, taint analysis is not a cure-all technique. Certain aspects of computing are overseen by it. Also, pure taint analysis does not suffice; often, extra techniques have to be implemented. To be able to target polymorphic attacks for example, extra effort has to be put into identifying the vulnerability instead of the exploit.

4.2.1 What is not taken into account

Not captured by taint analysis techniques are things like:

<table>
<thead>
<tr>
<th>Slice size</th>
<th>Requests/sec</th>
<th>Number of servers</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.34</td>
<td>148 (27%)</td>
<td>15</td>
</tr>
<tr>
<td>5.24</td>
<td>333 (62%)</td>
<td>30</td>
</tr>
<tr>
<td>0.25</td>
<td>380 (70%)</td>
<td>635</td>
</tr>
<tr>
<td>0.14</td>
<td>497 (92%)</td>
<td>1135</td>
</tr>
<tr>
<td>0.04</td>
<td>471 (87%)</td>
<td>3973</td>
</tr>
<tr>
<td>0.01</td>
<td>506 (94%)</td>
<td>15893</td>
</tr>
</tbody>
</table>

Table 4.1: Work-time quantums and their effects on Apache performance and AC size (taken from [22])
Privacy aspects of the user. If a spyware program somehow gets installed on the user’s machine, having the whole system run inside some taint analysing VM will not stop the program from listening to keystrokes and taking occasional screenshots, for example.

Non-control-flow violations are overseen by taint analysis. These could also be a buffer overflow, like with an information leak, but not caused by an operation on tainted data, but because ‘private’ memory contents are erroneously put on the wire.

Situations where the program counter walks into a block of tainted data on execution. The system will not notice this (no copying to the EIP\(^2\) register for example). However, processes generally perform a jump (a JMP, or a CALL, RET or BNE) about every 7 instructions\(^19\), at which point it will be detected, or at least risk will be less.

Higher level attacks e.g., macros, which cause certain complex behaviour that is not caught by tracking the taint flow.

\(^2\)Extended Instruction Pointer
Chapter 5

Signatures

Automatic tools for exploit analysis and signature generation are needed: as fine-grained detectors are expensive and thus may not be deployed on every vulnerable host, once a new exploit attack is detected, it is desirable to generate faster filters that can be widely deployed to filter out exploit requests before they reach vulnerable hosts/programs. An important mechanism playing a role here is content-based filtering, where content-based signatures are used to pattern-match packet payloads to determine whether they are a particular attack. Content-based filtering is widely used in intrusion detection systems such as Snort [32], Bro [27], and Cisco’s NBAR system [39], and has been shown to be more effective than other mechanisms, such as source-based filtering for worm quarantine [24]. However, these systems all use manually maintained databases of signatures, making for slow reactions. Manual signature generation is clearly too slow to react to a worm that infects hundreds of thousands of machines in a matter of hours or minutes. We need to have automatic exploit analysis and signature generation to quickly generate signatures for attack filtering after an exploit attack has been detected [26]. However, even if automatically generated and distributed, pattern-based signatures are useless against polymorphic worms, as the network packages change too.

Using taint analysis, a fingerprint of an exploit can be generated, after which several patches can be created and tested against the recorded attack automatically.

Generating signatures is useful for providing the rules to be used by the various filter services. However, patching the software so the exploit(s) are not going to have any effect anymore is another way of protection. Strictly speaking, a patch is a cure of some kind that makes the buggy code part invulnerable for exploits. This can be implemented by low-level filters, or by changing the code itself. The latter can be done by binary rewriting [31], or – if you have the source code – by changing the source and recompiling (the relevant part). Effectiveness of the various protection schemes depend on the speed and accuracy of deploying and generating signatures and corresponding patches.

5.1 Vigilante

Vigilante [6] generates so-called self-certifying alerts (SCAs). These signatures are then sent to other hosts: it “relies on collaborative worm detection at end hosts, but does not require hosts to trust each other. Hosts run instrumented software to detect worms and broadcast self-certifying alerts (SCAs) upon worm detection. SCAs are proofs of vulnerability that can be inexpensively
verified by any vulnerable host. When hosts receive an SCA, they generate filters that block infection by analysing the SCA-guided execution of the vulnerable software² ⁶. Vigilante uses a secure Pastry overlay ⁴ (a peer-to-peer network) to broadcast the SCAs.

The group has implemented a prototype of Vigilante for x86 machines running Windows and tested it against well-known worms: Slammer, CodeRed and Blaster.

5.1.1 Dynamic dataflow analysis

The base of the analysis Vigilante performs, is dynamic dataflow analysis. It is implemented using binary re-writing at load time, with instrumented versions of all instructions involved with moving of data. These are used to maintain data structures that not only taint memory blocks and registers, but also keep track of where the data came from. Every tainted register and memory location has an integer identifying the input message and offset where the dirty data originated, with the identifiers simply being a sequence number for every byte received in input messages. Also, a list with starting sequence numbers for every input message is kept to map identifiers to messages. The detector uses the data structures to generate an SCA on an infection attempt. From this SCA, the messages can be replayed, and the execution path be followed, enabling hosts to generate a filter. ⁶

5.1.2 Self-Certifying Alerts

Being a machine-verifiable proof of a vulnerability, an SCA proves the existence in a service of that vulnerability. Verifying SCAs is inexpensive, which is needed for fast action against worms. Applying an SCA to filter against malicious traffic is basically putting a patch into place. The filter (‘patch’) is generated automatically using dynamic data flow and control flow analysis of the execution path the specific worm follows, as described in the SCA.

Alert types

Vigilante uses three types of alerts. All three contain a sequence of messages that would cause a vulnerable service to reach a disallowed state. Also included is an identification of the vulnerable service, of the alert type and verification information for helping with the verification of the alert (specifying where to put the address of the code to execute in the sequence of messages for example). Verification is done by sending those messages to the service and checking whether that state happens, using detection engines and message logging. By using diverse detection engines, a reduction of the false negative rate can be accomplished. The three types of alerts are:

Arbitrary Execution Control alerts identify vulnerabilities that allow worms to redirect execution to arbitrary pieces of code in a service’s address space. They describe how to invoke a piece of code whose address is supplied in a message sent to the vulnerable service ⁶. The additional verification information specifies where to put the address of the code to execute in the sequence of messages – e.g., in which message and at which offset.

Arbitrary Code Execution alerts describe code-injection vulnerabilities. They describe how to execute an arbitrary piece of code that is supplied in a message sent to the vulnerable service
The verification information specifies where in the sequence of messages to place the code to execute.

**Arbitrary Function Argument** alerts identify data-injection vulnerabilities that allow worms to change the value of arguments to critical functions, for example, to change the name of the executable to run in an invocation of the `exec` system call. They describe how to invoke a specified critical function with an argument value that is supplied in a message sent to the vulnerable service [6]. Information about a critical function, a critical formal argument to that function, and where to put the corresponding actual argument value in the sequence of messages is included in the SCA.

**Alert verification**

As said, verifying an SCA is done by reproducing the infection process. This involves sending the sequence of messages in the alert to the vulnerable service. Vigilante’s aim is to have fast verification, to compete with the speed of the worms. Because SCAs may be originating from untrusted sources, the current implementation uses a separate virtual machine to contain any malicious side effects while verifying the exploit. To be able to reproduce correctly, the hosts have to use the same configuration to run the production instance of a service and the sand-boxed instance, as some vulnerabilities can only be exploited in certain program configurations.

Hosts run this virtual machine with instrumented versions of network-facing services and a verification manager. The instrumented services signal the manager if the exploit is verified. Also, an SCA verifier runs outside the virtual machine, providing other processes with an interface to the verification module and acting as a reverse firewall.

This verification scheme is fast, simple and generic and has no false positives. The time it takes to verify an SCA is similar to the time it takes the worm to infect the service, as the overhead of the instrumentation and virtual machine are small. There is always a virtual machine kept ready to verify SCAs, ensuring timely distribution of alerts. Verification is independent of the detection engine used to generate the alert, as the SCAs are standardised. This is important for keeping the trusted computing base small, especially with many distinct detectors running in the system. Also, if the verification procedure signals success, the service is vulnerable to the exploit described in the SCA, showing attackers can control this vulnerable service through its external messaging interface.

Vigilante uses host-based filters to block worm traffic before it is delivered to the vulnerable service. These filters do not change the vulnerable service and they allow the service to continue running under attack. Shield [44] also proposes host-based filters, but Vigilante describes a mechanism to automate generation such that they have no false positives, are effective at blocking worm traffic, and introduce very low overhead.

### 5.2 Argos

Argos differs from Vigilante in the field where it supports both virtual and physical addresses, and has access to the whole system whereas Vigilante only supports virtual addresses and monitors single processes [31]. This enables Argos to dump more information about the system to file besides the tainted blocks and addresses, like registers, physical memory blocks and specific
virtual addresses and all its mappings. Because it marks the address triggering the violation, and has information to be able to translate between the virtual address and the physical address, it now can generate signatures, and has plenty of evidence for manual analysis as well.

When an attack is detected, it may not be known yet which process causes the alarm. To get more information about the application, like the process identifier, executable name, open files and sockets, Argos injects its own OS-specific shell-code to perform forensics, actually sort of exploiting the code under attack with their own shell-code. This technique is not mandatory, but provides a higher accuracy.

It then can correlate the dump of memory blocks (with tainted data, registers, etc.), additional information obtained with the shell-code and the network traces in the trace database to generate a real signature. In case of TCP connections, the reconstruct flows prior to correlation [31]. However, this signature is not the final yet, as it can still be refined.

5.2.1 SweetBait

For refining signatures, Argos uses a subsystem called SweetBait, which correlates signatures from different sites, and refines signatures based on similarity [30]. SweetBait can for example notice the resemblance between two signatures of the same attack on different hosts, where the differing IP addresses lead to different signatures. It can then generate a shorter, more specialised signature that is then used in subsequent filtering [31]. SweetBait also functions as deployment system for the signatures [30], and can collect snort-like signatures from various honeypots – like honeyd – with Argos being the largest. Honeyd is a honeypot using scripts emulating existing services to gather information on attacks. It uses ScriptGen [21] to generate those scripts, as they are hard to write by hand, and rare for proprietary protocols. Argos uses a control centre (CC) to maintain a database of attack signatures, pushing them to a variety of sensors. Honeypots can send new signatures to the CC over an SSL connection, after which the CC will compare it to its database and may decide that it is a more refined version of an existing one, installing it as an updated signature. This way, SweetBait tries to eliminate the parts that vary, like the IP addresses, attack payloads, host names etc.

The control centre takes care of gathering and refining signatures, and pushing them back to remote intrusion detection and prevention systems (IDS and IPS) [31].

5.3 Prospector

Prospector is build on top of Argos and targets polymorphic network attacks [36]. Most signature generating systems look for simple byte patterns. However, exploits can be polymorphic [13]: a worm can change its code and attack payload (basically, its signature) by including bogus data and shuffling parts around. Prospector tries to fingerprint the vulnerability instead of creating a signature of the exploit(s) attacking it. To be able to do this, a simple question has to be asked: “what bytes contribute to an attack?”. It appears the answer is quite hard though [36]. However, an answer leads to easy generation of signatures that meet the specifications of being able to cope with polymorphism (essentially targeting the vulnerability).

Prospector is implemented and evaluated on Linux and uses an emulator-based honeypot with dynamic taint analysis (Argos) to detect the attacks and to locate both the exact address where control flow diversion occurs and all the memory blocks that originate in the network
Taint Analysis in Practice

(tainted bytes). Its techniques apply to other operating systems too. The signature generator is based on Ethereal\textsuperscript{1} but can be easily ported to any other network protocol analyser.

```c
11  void read_from_socket (int fd) { // fd is the socket descriptor
12         int n; // the vulnerable buffer
13         char vuln_buf [8]; // a safe buffer, unrelated to the attack
14         char unrelated [8]; // from socket: taints all data in ‘vuln_buf’ and
15         // above (overflow possible)
16         read (vuln_buf, fd, 32); // from socket: taints all data in ‘unrelated’
17         // (no overflow possible)
18         read (unrelated, fd, 8); // untaints 4 bytes of data that was previously
19                // tainted, creating a gap
20         n = 1;
21         return;
22 }
```

Figure 5.1: Tainted data: gaps and dirt (unrelated tainted data) (taken from \cite{36})

Tracing which exact bytes are related to the vulnerability and weeding out the noise is not easy. When investigating a heap or stack overflow (see Section \ref{sec:overview} for more information), one way to get tainted data leading to this exploit is back-tracing from that point (the memory area beneath the violation address reported by Argos). As can be seen in Figure 5.1, simply back-tracing tainted buffers can include irrelevant data (like the unrelated one, which Prospector calls dirt), or not be detected because they where untainted (the variable \texttt{n}); Prospector calls this a gap. Prospector uses the age of the tainted data to determine whether it is relevant for the attack, or just a left-over from an older allocation.

Data that is used for the overflow may be originating from more than one set of bytes coming in from the network, so once Prospector knows which bytes where possibly relevant, it can easily find out which protocol fields contributed to the attack: if \texttt{n} fields were involved in the overflow with a combined length of \( N \), we know that any similar protocol message with a combined length for these field greater or equal to \( N \) will also lead to a buffer overflow \cite{36}. This method is not sufficient for countering attacks based on messages that contain a specially crafted (wrong) length field, misspecifying the length of another field. To detect such attacks, the signature pinpoints the length field and specifies when misbehaviour occurs.

Pinpointing which bytes contribute to an overflow is one of the major goals of Prospector. The project’s main contributions are:

1. accurate pin-pointing of bytes in heap or stack overows (and double frees attacks);
2. accurate tracing of such bytes to protocol elds in the network trace;
3. accurate signature generation for polymorphic exploits.

In case of advanced heap overflows, some more work has to be put into finding the vulnerable buffer. It requires first finding the memory region containing that buffer, then starting the marking of bytes contributing to the attack. It could be that at detection time, that memory region is reused and contains unrelated data. The emulator therefore marks the bytes surrounding an allocated chunk of memory as red. If tainted data is written in such a region – indicating an overflow but not necessarily an attack – the application is kept running, but the whole memory region is dumped for potential later use. In the case of an intrusion attempt, the dumped heap

\textsuperscript{1}Ethereal is nowadays known as Wireshark, \url{http://www.wireshark.org/}

24
areas are searched for the violation address and the network index to find the memory region
containing the buffer that contributed to the attack. On these pieces, further analysis is done to
overcome the before-mentioned difficulties. Because tainted memory is dumped exactly at the
moment of overflow, it is unlikely that data is omitted. The red markers are vaguely similar to
StackGuard’s canary values [7]. However, they differ in that they are maintained by the emulator
and trigger action immediately when they are overwritten.

The final signature consists of a sequence of value fields – specifying the specific value of
a field in the protocol – and critical fields \(^2\) which collectively satisfy some condition, like for
instance collectively having a length that is less/not less than \(L\). \(^3\)

The ‘red’ memory chunks can become stale after a while, so Prospector keeps counters in-
dicating the number of red markers associated with each physical page in memory, and a table
of physical pages associated with the identifier of the last process using the TLB \(^3\). Whenever a
new entry corresponding to a kernel address or the user stack is added to the TLB buffer, the
page is checked for having a new owner. If it has, any ‘red’ markers are removed.

Other contributions include a novel way to monitor process switches from an underlying
emulator and an attack vector-specific extension to Prospector to make it deal with double-free
attacks.

Performance is about 20% slower than pure Argos. The emulation, taint analysis and age
tracking are all responsible for a fair amount of overhead, but the authors state that Prospector
is well-suited for honeypots.

\(^2\)A critical field is a protocol field containing either the violation address, or includes at least one boundary
that can be mapped to the indicated malicious data
\(^3\)Translation Lookaside Buffer (TLB) is a CPU cache that is used by memory management hardware to improve
the speed of virtual address translation
Chapter 6

Conclusion

As long as software is being written in languages susceptible to memory bugs – like C – these vulnerabilities will not cease to exist. The sheer speed and hardware control these languages provide to the programmer are just too convenient. As long as there are humans involved in programming with these languages, errors in memory management will exist; protecting software with a sort of supervising mechanism keeping its eyes on the buffers only seems logical.

Taint analysis is a really convenient way of intercepting abuse of the vulnerabilities in the code. All implementations suffer from a performance hit, which can be alleviated a bit with hardware support, consisting of memory and CPU extensions for taint bookkeeping. For situations where speed counts, it seems practical to combine a taint analysis system with a system with a lower overhead, like address space randomisation for generating the exceptions triggering further investigation. This investigation can be done on shadow servers, running instrumented versions of the services, or by ‘switching on’ the instrumentation of the already running process on the main server. Alex Ho et al. as well as a recent version of Argos for example make a good point with running the system in a Xen virtual machine (which causes only a slight overhead), and providing a mechanism to enable and disable real taint analysis instrumentation at will.

Because of the generally high overhead, it seems more logical to implement such systems on important big-iron servers than on the desktop, where the user cares more about speedy execution than safety in general. However, seeing how many desktops are being targeted by worms, causing hundreds of thousands of machines to turn into zombies swamping the Internet with spam and other unwanted material and spying on their owners, it seems called for to at least implement a rudimental system on consumer machines too. For example, running web browsers and other known buggy software like certain instant messaging programs in an instrumented sandbox might protect against a lot of modern worms and spyware.

With worms and other malware evolving, it could very well be that one day not only integer registers and the like have to be monitored, but also floating point and MMX instructions. Multimedia programs and games use those kinds of instructions for doing their job, and have been using information from the Internet for years; being it a video stream or a connection to the server for multi-playing. If malicious information is loaded into those registers and pipelines, it might just as well lead to a buffer overflow or information leak. Only future will tell.

Of course, taint analysis is no holy grail of protection. It will not stop a spyware program listening to your keystrokes or popping up advertisements of course.
Bibliography


Taint Analysis in Practice


