

Fast and Black-box Exploit Detection and Signature Generation for Commodity Software

XIAOFENG WANG, ZHUOWEI LI and JONG YOUL CHOI

Indiana University

JUN XU

Google Inc. and North Carolina State University

MICHAEL K. REITER

University of North Carolina at Chapel Hill

and

CHONGKYUNG KIL

North Carolina State University

In biology, a *vaccine* is a weakened strain of a virus or bacterium that is intentionally injected into the body for the purpose of stimulating antibody production. Inspired by this idea, we propose a *packet vaccine* mechanism that randomizes address-like strings in packet payloads to carry out fast exploit detection and signature generation. An exploit with a randomized jump address behaves like a vaccine: it will likely cause an exception in a vulnerable program's process when attempting to hijack the control flow, and thereby expose itself. Taking that exploit as a template, our signature generator creates a set of new vaccines to probe the program in an attempt to uncover the necessary conditions for the exploit to happen. A signature is built upon these conditions to shield the underlying vulnerability from further attacks. In this way, packet vaccine detects exploits and generates signatures in a black-box fashion, that is, not relying on the knowledge of a vulnerable program's source and binary code. Therefore, it even works on the commodity software obfuscated for the purpose of copyright protection. In addition, since our approach avoids the

11

This work is supported in part by the Cyber Trust program of the National Science Foundation under Grant No. CNS-0716292, and by I3P/Department of Homeland Security under Grant No. I3P/DHS 5-36423.5780 at Indiana University. Partial support is also provided by the National Science Foundation under Grants No. 0433540 and No. 0326472.

A preliminary version of this article appears in the *Proceedings of 2006 ACM Conference on Computer and Communications Security (CCS'06)* [Wang et al. 2006].

Authors' addresses: X. Wang, Z. Li, and J. Y. Choi, Indiana University; email: Zhuowei.li@microsoft.com; J. Xu, Google Inc. and North Carolina State University; M. K. Reiter, University of North Carolina at Chapel Hill; C. Kil, North Carolina State University.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credits is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from the Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2008 ACM 1094-9224/2008/12-ART11 \$5.00 DOI: 10.1145/1455518.1455523.

<http://doi.acm.org/10.1145/1455518.1455523>.

ACM Transactions on Information and System Security, Vol. 12, No. 2, Article 11, Pub. date: December 2008.

expense of tracking the program’s execution flow, it performs almost as fast as a normal run of the program and is capable of generating a signature of high quality within seconds or even subseconds. We present the design of the packet vaccine mechanism and an example of its application. We also describe our proof-of-concept implementation and the evaluation of our technique using real exploits.

Categories and Subject Descriptors: K.6.5 [**Security and Protection**]: Invasive software, Unauthorized access

General Terms: Security

Additional Key Words and Phrases: black-box defense, exploit detection, signature generation, worm, vaccine injection

ACM Reference Format:

Wang, X., Li, Z., Choi, J. Y., Xu, J., Reiter, M. K., and Kil, C. 2008. Fast and black-box exploit detection and signature generation for commodity software. *ACM Trans. Inf. Syst. Secur.* 12, 2, Article 11 (December 2008), 35 pages. DOI = 10.1145/1455518.1455523. <http://doi.acm.org/10.1145/1455518.1455523>.

1. INTRODUCTION

In biology, a vaccine is a living, weakened strain of a virus or bacterium that is intentionally injected into the body for the purpose of stimulating antibody production. That strain is weakened to prevent it from causing infection. Similarly, a “weakened” exploit packet with important elements of its payload scrambled would quickly expose itself through the exception it causes in a vulnerable program. Forensic analysis of the exception could uncover the related program vulnerability and enable the generation of an “immunity,” a signature for capturing future exploits on the same vulnerability.

The above intuition can be applied to exploit detection, vulnerability diagnosis and automatic signature generation. Design of such mechanisms has been impeded by the constraints of commodity software, for which access to source or binary recompilation is often prohibited. Existing approaches [Newsome and Song 2005; Crandall et al. 2005; Costa et al. 2005] have suggested tracking the input data as the program executes until the point at which control-flow hijacking happens. We call these approaches *gray-box analysis*, as they do not need source code (as a *white-box* approach would) but do have to monitor a program’s execution flow closely (a *black-box* approach would not). Gray-box analysis is accurate and applicable to commodity software; however it incurs significant runtime overheads, often slowing the system by an order of magnitude.

Inspired by the principle of vaccination, we develop a much faster black-box approach. Rather than using expensive dataflow tracking, it detects and analyzes an exploit using the outputs of a vulnerable program. Specifically, we first identify anomalous tokens in packet payloads, for example, byte strings resembling injected jump addresses in a control-flow hijacking attack, and randomize the contents of these tokens to generate a vaccine. If the packets carrying these tokens indeed contain an exploit, the vaccine will likely cause an exception in the vulnerable software. When this happens, our approach will automatically generate a signature to protect the software using the forensic

data gleaned from the exception and fault injection techniques [Musa et al. 1996]. We call this approach packet vaccine.

1.1 Contributions

Compared with other techniques, packet vaccine offers some important benefits:

1.1.1 *Fast, black-box exploit detection.* Packet vaccine detects an exploit attempt by directly injecting vaccine packets into a program. Therefore, it performs as fast as a normal run of that program, and up to an order of magnitude faster than gray-box approaches. In addition, packet vaccine does not use source code or recompiled binaries and thereby works well with commodity software.

1.1.2 *Effective signature generation.* Packet vaccine generates signatures using host information, so it is immune to interference from Internet noise [Richardson et al. 2005] and poisoning [Perdisci et al. 2006], which can mislead network-based signature generators (e.g., Early Bird [Singh et al. 2004], Polygraph [Newsome et al. 2005], Nemean [Yegneswaran et al. 2005]) into generating false signatures. Moreover, the resulting signature tends to capture some key properties of a vulnerability such as the size of a vulnerable buffer, which can be used to detect a range of exploit mutations employed by polymorphic worms.

Using a confirmed exploit as a template, packet vaccine can generate a number of vaccines, that is, variations of that exploit, to gain a better characterization of a software application’s vulnerability. For instance, one type of our signatures uses a packet’s field length as an attribute to identify a buffer-overflow attack; injection of vaccines with different field lengths allows us to accurately estimate the size of the underlying vulnerable buffer and thereby generate a more accurate signature (Section 2.3). Moreover, our technique can generate a signature without any information about an application or its protocol.

Some gray-box approaches perform static analysis [Brumley et al. 2006; Newsome et al. 2005] over a vulnerable program’s binary code and could generate signatures more accurate than our signatures. However, our black-box approach tends to be faster than those approaches and even works with obfuscated code [van Oorschot 2003; Naumovich and Memon 2003]. For many exploits, our black-box technique can produce signatures close to their signatures in quality, as we report in our experimental study. We argue that a rapidly-generated and reasonably accurate signature could be more useful in practice because such a signature is supposed to serve as a temporary defense pending the release of a patch, rather than a permanent fix [News 2006].

1.1.3 *Low overhead and easy deployment.* Packet vaccine is more lightweight and easier to deploy than many existing techniques. Exploit detection using our approach does not require installing anything on the host running vulnerable programs. Vulnerability diagnosis needs only a lightweight collec-

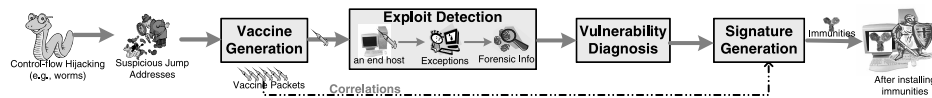


Fig. 1. The design of packet vaccine.

tor to gather forensic data from an exception, and even this requirement can be waived for operating systems which already offer error logging and debugging services. For example, Windows XP's event logs contain everything we need, such as corrupted pointer contents.

1.2 Application Domains

The current design of packet vaccine can detect only exploits carrying jump addresses for hijacking a vulnerable program's control flow. This limitation, however, does not apply to our signature generation techniques: once an exploit has been captured (possibly by other approaches such as Vigilante [Costa et al. 2005]), our techniques could be used to generate a signature for the underlying software vulnerability even if the exploit does not carry any jump addresses. As an example, ShieldGen [Cui et al. 2007], a recent proposal following the same idea as our *application-based vaccine injection* (Section 2.3), was shown to work successfully on the WMF vulnerability [US-CERT] which is exploited without injecting jump addresses.

So far, we have used packet vaccine to protect only stateless services such as static Web servers, as described in Section 4. It remains to see how our techniques can be adapted to protect stateful services such as FTP.

1.3 Roadmap

We present the design of the packet vaccine mechanism (Section 2) and the implementation of this technique in the article. We evaluate it using real exploits and signatures generated by a gray-box approach (Section 3). Our study shows that packet vaccine can effectively detect exploits, and efficiently generate signatures of high quality. To apply this technique to protect an online service, we present an architecture which employs test servers to carry out exploit detection, and empirically evaluate its performance with a proof-of-concept implementation (Section 4). We also discuss other potential applications and the limitations of our approach (Section 5), review related work (Section 6), and outline future research (Section 7).

2. DESIGN

In this section, we present the design of the packet vaccine mechanism. Figure 1 illustrates the major steps of our approach: *vaccine generation*, *exploit detection*, *vulnerability diagnosis*, and *signature generation*.

Vaccine generation is based upon detection of anomalous packet payloads, for example, a byte sequence resembling a jump address, and randomization of its contents. A vaccine generated in this way serves to detect an exploit attempt which, if present, could trigger an exception from a vulnerable

program. Vulnerability diagnosis correlates the exception with the vaccine to acquire some key information regarding the exploit, in particular the corrupted pointer content and its location in the exploit packet. Using this information, the signature generation engine creates variations of the original exploit to probe the vulnerable program, which could identify necessary exploit conditions for generation of a signature.

2.1 Vaccine Generation

To generate a vaccine, we need to preserve the exploit semantics while weakening it enough to prevent a control-flow hijacking from taking place. Here, we describe a simple way to do that.

A key step in most exploits is to inject a jump address to redirect the control flow of a vulnerable program. Such an address points to somewhere in the stack or heap in a code-injection attack, or to a global library entry in an existing-code attack. Our approach is to check every 4-byte sequence (32-bit system) or 8-byte sequence (64-bit system) in a packet’s application payload, and then randomize those which fall in the address range of the potential jump targets in a protected program. The vaccine generated in this way should cause an exception, segmentation fault (SIGSEGV), or illegal instruction fault (SIGILL) to a vulnerable program’s process if an exploit is indeed present in the original packet. A question here is how to determine the address range.

2.1.1 Address Range. A process’s virtual memory layout is usually easy to obtain. On Linux and UNIX, the `proc` virtual filesystem maintains a file called `maps` under the directory `/proc/pid/` which offers the runtime memory layout for the process `pid`. From that file, we can obtain the base addresses for the stack (usually from `0xc0000000` downwards) and the entry for function libraries (in segment `0x40000000`). The base address for heap is the end of BSS which can be determined by analyzing the binary executable using tools such as `objdump` or `readelf`. To find out the address range, we also need to know an application’s stack and heap sizes. These can be estimated by monitoring stack and heap usage recorded in the “status” file of the application’s process for a period of time. Another source for gauging heap’s usage is the packet size of a request to the application. Using these data, we determine the address ranges as follow. Let b_s and u_s be the stack’s base address and usage respectively. Stack addresses are estimated to range from $b_s - \alpha u_s$ to b_s , where $\alpha \geq 1$ is a ratio for keeping a safe margin. Similarly, heap range is approximated as b_h to $b_h + \alpha u_h$, where b_h and u_h are the heap’s base and usage respectively.¹ Address ranges can also be customized by the user. For example, one could restrict monitoring to the heap on an operating system with a nonexecutable stack.

We can pinpoint the address range of the global libraries intensively used by exploits, for example, `msvcrt.dll` or `libc.so`, and even the entry addresses of some “dangerous” functions such as `system()` and `execve()`. These ad-

¹A process may have multiple heap regions, which can be observed from its memory maps. In this case, we can use the base addresses of these regions plus αu_h to estimate multiple heap address ranges.

addresses can be easily acquired on Linux or UNIX using the `maps` file and the command `nm`. A Windows application’s memory information can be collected using memory monitoring tools like `Memview` [MemView 2006] or debugging tools such as `CDB` or `NTSD` [Microsoft 2007]. The address range could also cover the global offset table (GOT), though this might not be necessary: an exploit usually changes a function pointer in the GOT to an address in the stack or heap, where the attack code lies. Again, it is at the user’s discretion to decide the coverage of the address range. The larger the range becomes, the more packets must be checked and randomized.

Address ranges can also be approximated through an empirical study of known exploits, which could reveal the “hotspots” to which most exploits jump. In our research, we collected around 1,000 jump addresses from known exploits and discovered that on Linux, most code-injection attacks use the jump addresses either in the range `0xbfff0000` to `0xbfffffff` for the stack or `0x08040000` to `0x08ffffff` for the heap. This treatment also works for existing-code attacks, as most of these exploits use a small set of `libc` (Linux or UNIX) or `dll` (Windows) functions as stepping stones. We present the distribution of exploit addresses in Appendix A.

2.1.2 Vaccine Generation Algorithm. Now we are ready to present the vaccine generation algorithm, which is formally described as follows.

- Gather data from the application being protected and build a *target address set* $T = [b_s - \alpha u_s, b_s] \cup [b_h, b_h + \alpha u_h] \cup S$, where S is a set containing the address ranges of objects other than the stack and heap, such as the entries for global library functions.
- Aggregate the application payloads of the packets in one session into a dataflow, carry out a proper decoding (e.g., Unicode decoding, URL decoding, etc.) if necessary and scan that dataflow to find all byte sequences $\tau \in T$.
- For every τ , replace its most significant byte with a byte randomly drawn from a scrambler set R to output a new dataflow.
- Construct vaccine packets using the new dataflow as application payloads.

In the above algorithm, the scrambler set R could be set to avoid introducing some undesired symbols (such as syntax tokens) which could interrupt a protocol, and ensure a randomized byte sequence falls outside a process’s memory map. An example of R is $\{A \text{ to } Z, a \text{ to } z, 0 \text{ to } 9, “+” \text{ and } “-”\}$.

For example, the payload of Code Red II worm is presented in Figure 2. Our vaccine generator identifies multiple occurrences of the byte sequence “`0x7801cbd3`” from the payload after Unicode decoding. This sequence falls in the address range of `msvcrt.dll`, which is being monitored. Therefore, a vaccine is generated as illustrated in Figure 2, in which the most significant bytes of the sequence have been scrambled.

2.1.3 Discussion. A central question here is whether the vaccine generated above is effective in detecting an exploit if it is indeed present. Exploits tend to be fragile, in the sense that a random perturbation could cause them to vanish. For example, randomization of protocol syntax tokens, such as the

to manipulate. However, our approach is subject to evasion if an application indeed has a vulnerability which allows for an exploit with an address-like semantic token.

Our randomization strategy also helps preserve exploit semantic tokens: instead of scrambling the whole byte sequence, we only modify one byte, namely the most significant. We could extend this idea, for example, by generating three vaccines, each of which scrambles one of the three most significant bytes on the sequence. These vaccines can be used to probe an application in parallel. As a result, even if an exploit does have an address-like two-byte string (such as `0xbfff`) on which it depends, we can still detect it. Another approach involves a simple network anomaly detector (NAD) which narrows down the search for address-like substrings to only part of an anomaly packet's payload. For example, a NAD monitoring the length of packets' application fields may identify an overlong CGI parameter in a Code Red II packet; this allows a vaccine generator to only scan that field, avoiding randomizing the semantic token “.ida?” even if it does look like an address (which it does not in reality).

2.2 Exploit Detection and Vulnerability Diagnosis

Exploit attempts from vaccine packets are detected from the exceptions they cause in a vulnerable program, such as `SIGSEGV` and `SIGILL`. Such exceptions happen with an overwhelming probability if exploits' jump addresses have been scrambled.

The objective of vulnerability diagnosis is to reliably correlate an exception with one of the byte sequences being randomized, which identifies the location of the jump address on an exploit packet. This correlation is established by matching these byte sequences to the forensic data gathered from an exception, in which the corrupted pointer is of particular importance. On x86 systems, the corrupted pointer which causes a `SIGSEGV` exception can be found in register `CR2`. It may also appear in register `EIP`. Our approach logs the contents of these registers once an exception happens.

Formally, vulnerability diagnosis works as follows. Let $\tau_1, \tau_2, \dots, \tau_n$ be a set of n byte sequences (tokens) on a vaccine packet which have been scrambled by the vaccine generator. Let p be the forensic string, i.e., the corrupted pointer collected from registers. If $p = \tau_i$ for $1 \leq i \leq n$, we correlate τ_i with the exception. This correlation can be validated using the following test: we randomize all bytes on τ_i to produce a new token τ and use it to generate a new vaccine; sending this vaccine to the vulnerable program, we check whether the exception happens again and the corrupted pointer also changes to τ . The validation test can be repeated to increase the confidence in the correlation.

Liang et al. propose `COVERS` [Liang and Sekar 2005b] which detects an exploit attempt using address space randomization (ASR) and identifies the exploit packet by correlating its content with the memory range around a corrupted pointer. By comparison, the correlation offered by our approach is more reliable. `COVERS` could be subject to a miscorrelation attack in which an intelligent attacker duplicates memory data collected from an unrelated exception into a packet which carries no exploit payload. As a result, a false signature

could be generated to filter out legitimate packets. Such an attack does not work on packet vaccine, as we identify an exploit packet using the byte sequence scrambled by the vaccine generator. In order to establish a strong correlation, COVERS needs to find the content of the entire input buffer. Given the diversity of exploits, this may have to be done on a case-by-case basis. By comparison, packet vaccine performs validation tests and can therefore simply trust registers' contents, which are easier to collect. Finally, ASR could cause an exploit to execute some legitimate instructions before it triggers an exception. When this happens, a correlation cannot be established. Our approach could avoid this problem: we can randomize an address-like byte sequence to an invalid address, outside a process's memory map.

2.2.1 Resilience to Metamorphism and Polymorphism. A significant challenge to exploit detection is polymorphism and metamorphism, which transform the instructions in exploit instances to evade detection based on the prevalence of these instructions. These attacks, however, can be defeated by packet vaccine in some cases, as we explain as follows.

An exploit packet usually consists of a segment of jump addresses, a segment of instructions and a segment of nop-like padding. Metamorphism and polymorphism can only be applied to the instructions and the padding, as the jump addresses serve as a critical entry point to hijacking the control flow of a running program and are therefore indispensable to a successful exploit. Previous research shows that different exploit instances vary in only the least significant byte of their jump addresses [Newsome et al. 2005; Pasupulati et al. 2004; Newsome et al. 2005]. This does not prevent packet vaccine from detecting them because they must be all inside the address ranges of potential jump targets.

2.3 Signature Generation Using Vaccines

After vulnerability diagnosis, we have identified the jump address and its location in an exploit packet. The address alone, however, could be too general to be a signature, especially for binary protocols such as DNS. More information is required to form a high-quality signature. Here, we describe a *signature generation engine* which uses a known exploit as a template to generate faults (vaccines) and injects them into a vulnerable program to acquire key attributes of the underlying vulnerability. We call this technique *vaccine-injection* (VI). Our approach can generate signatures with or without the application-level information, as we elaborate below.

2.3.1 Application-Independent Signature Generation. We can generate a signature without any knowledge about an application's protocol. Such a signature is in the form of a *token sequence*, which consists of an ordered sequence of byte strings (tokens) [Newsome et al. 2005]. These tokens' locations on the exploit packet's payload could also be included as a part of the signature for a binary application protocol such as DNS. Our idea is to determine the roles played by individual bytes in an exploit by scrambling them to create vaccines

and testing them in the vulnerable application, in an effort to identify the inputs necessary for the exploit to occur.

Let L be the byte length of an exploit dataflow, and $B[i]$ be the i th byte on that dataflow, where $1 \leq i \leq L$. Suppose the scrambled jump address τ with a byte length l starts from the r th byte. The signature generation engine generates $L - l$ vaccines, $\{v_1, v_2, \dots, v_{r-1}, v_{r+l}, \dots, v_L\}$, such that v_i ($i \in [1, r-1] \cup [r+l, L]$) randomizes the i th byte on the exploit payload and also keeps the token τ . Then, it injects all these vaccines into the vulnerable program. If v_i does *not* cause any exception, we record $B[i]$ (and also i for a binary protocol) as a token byte. All the contiguous token bytes are recorded as a token. A signature is formed using these tokens and the target address set T . A dataflow is deemed to match such a signature if it contains all these tokens and at least one byte sequence in T . We refer to this approach as *byte-based vaccine injection* (BVI).

Performance. Some servers process requests using multiple processes. Crashing one of their processes does not affect any of the others. This property allows us to test many vaccines in parallel. In addition, many exploits have a short exploit payload, usually below 1kB. In such cases, BVI offers good performance. We also adopted a *block-searching* technique to reduce the number of vaccines for generating a signature. We first test a vaccine which randomizes a block of contiguous bytes on an exploit packet. If the vaccine still causes the exception, we move on to randomize another byte block; otherwise, we test every byte inside that block to identify signature tokens.

Disjunctive Tokens. An attacker might duplicate a semantic token to several places. For example, the Code-Red II worm (Figure 2) has multiple “%u” tokens, but only one is necessary for the exploit to succeed. This prevents BVI from detecting that token, as randomization of one of its replicas does not make the exception disappear. We can mitigate this problem through the following improvement of the BVI algorithm, called *BVI-1*, which captures the last of the repeated tokens. A vaccine v'_i scrambles the first i bytes on the exploit dataflow except all the signature tokens having been identified so far. If the vaccine does not cause any exception to the vulnerable program, the signature engine records the i th byte as a new signature token. Otherwise, our approach scrambles that byte before generating the next vaccine v'_{i+1} . A performance problem of BVI-1 is that it cannot be parallelized.

Both BVI and BVI-1 cannot identify all the tokens of disjunction relations, for example, such that any of these tokens suffices for the exploit to occur. To solve this problem, we developed another algorithm, called *BVI-2*, which is presented below. Here, we use *disjunctive set* to describe a set of tokens with a disjunctive relation. The set is active if some of its members are yet to be identified. In addition, the jump addresses always remain scrambled in all steps of the algorithm.

- (1) Run the BVI algorithm to identify all the tokens except those with disjunctive relations.
- (2) Detect the existence of disjunctive tokens as follows:

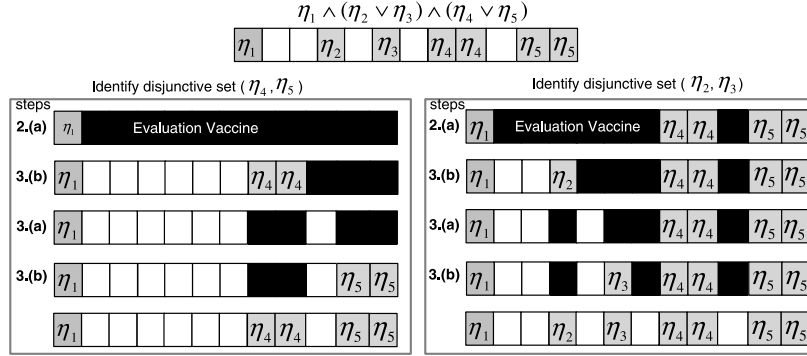


Fig. 3. An example for identifying conjunctive and disjunctive tokens.

- (a) Generate an *evaluation* vaccine which keeps all the identified tokens intact, and scrambles all other bytes including the jump addresses on the application payload of an exploit packet. Inject that vaccine into the vulnerable program.
 - (b) If the vaccine *causes* an exception, declare there is no disjunctive tokens unidentified and goto Step 5.
- (3) Create a new, empty disjunctive set, set it as *active* and run the following algorithm to find the members of that set. Here λ is the total number of scrambled bytes on the evaluation vaccine.
- For $i = 1$ to λ do
- (a) Generate a *test* vaccine which restores the first i scrambled bytes on the evaluation vaccine but scrambles the tokens already in the active disjunctive set, and inject it into the vulnerable program.
 - (b) If there is an exception, we know the i th byte is the last byte of a token, and proceed as follows to find other bytes of that token and save it:
 - i Let m be the number of restored bytes on the test vaccine. $m < i$ if the first i bytes include tokens inside the active disjunctive set. Generate m vaccines, each of which scrambles exactly one of the restored bytes on the test vaccine, and inject them into the program.
 - ii If a vaccine does not cause an exception, save the scrambled byte on it as a token byte. All the token bytes form a token.
 - iii Save the token into the active disjunctive set.
 Set the active disjunctive set as inactive.
- (4) Go to Step 2.
- (5) Output a regular-expression signature composed of a conjunction of all the tokens outside disjunctive sets, and disjunctions of the tokens within the same disjunctive sets.

Figure 3 offers an illustration of this algorithm. Suppose an exploit packet contains five tokens: η_1 to η_5 , among which (η_2, η_3) and (η_4, η_5) are of disjunctive relations respectively, as described in the figure. η_4 and η_5 both have two bytes. Other tokens have only one byte each. Step 1 of the algorithm only

identifies η_1 , because scrambling one byte on any other tokens does not make the exception disappear. In Step 2, the evaluation vaccine does not cause an exception, which suggests the existence of disjunctive tokens. The “for” loop in Step 3 produces the first test vaccine which causes an exception when all the bytes of η_4 and other bytes prior to it are restored. Then, Step 3(b) detects the first byte of that token, creates a new active disjunctive set and save the token to that set. To proceed that loop, the identified token η_4 must always be scrambled, which enables the discovery of η_5 . After detecting both tokens, we run Step 2 again, which suggests the existence of other disjunctive sets. We repeat Step 3 to find another disjunctive pair (η_2, η_3) and go back to Step 2. This time, no disjunctive tokens are found. Therefore, the algorithm gets to Step 5 to output the signature $\eta_1 \wedge (\eta_2 \vee \eta_3) \wedge (\eta_4 \vee \eta_5)$.

Step 3(b) may generate a noncontiguous token, which can also be viewed as a conjunction of multiple contiguous tokens. The BVI-2 algorithm was evaluated in our experiment (Section 3) over the `rpc.statd` exploit which contains a disjunctive token “%n”. The weakness of the approach, however, is performance, which suffers from the extra rounds of tests. Fortunately, such a disjunction trick cannot be played on most tokens (e.g., “.ida” and “GET”) and thus the original BVI algorithm works in many cases.

Exploit-Specific Tokens. Our approach might generate a signature containing some tokens specific to an exploit. Taking Code Red II as an example, the signature engine will extract “.ida?” which is exploit-specific, as another exploit containing the token “.idq?” can attack the same vulnerability. This problem seems inevitable when generating signatures in the absence of application information. However, we argue that our approach is more efficient and accurate than other application-independent techniques such as network-based signature generation, as our signature is generated from a single exploit instance, has a negligible probability of false positives and removes all wildcard strings the attacker can use for a “red herring” attack [Newsome et al. 2005]. In addition, our technique runs faster and generates a better signature using multiple exploit instances (see Section 3.3), and can also be used to refine a signature generated by a network-based signature generator (Section 3.3).

2.3.2 Using Protocol Information. If an application’s protocol specifications are available, we can generate a very accurate signature, close to a vulnerability-based signature. Such a signature makes use of the characteristics of buffer-overflow exploits and format-string exploits to describe a vulnerability. These exploits constitute most known control-flow hijacking attacks. The algorithm for generating these signatures is also built upon the VI technique, and therefore we call the approach *application-based vaccine injection* (AVI).

Buffer-overflow exploits are usually characterized by anomalously long fields [Liang and Sekar 2005b]. Thus, a signature with the form (*application, command, field.name, max.field.size*) offers a good description of the vulnerability being exploited. Our signature generation engine first identifies the

application field which includes the jump address, and then makes a quick estimate of that field’s length using the number of the bytes prior to the address within that field. This gives a coarse signature. To refine that signature, our approach iteratively alters the field size to generate new vaccines, and injects them into the vulnerable program. If a vaccine makes the exception disappear, we know the field is too short and then increase it. Otherwise, we shrink that field. Using a binary searching algorithm, we can quickly determine the minimal length for the exploit to happen. The signature generated in this way can be pretty close to the size of a vulnerable buffer: for example, our experiment over ATP httpd [SecurityFocus 2006] produced a signature only 23 bytes² longer than the real size of the program’s vulnerable buffer.

Format-string exploits usually contain the special symbol “%n”. In addition, the address token usually appears prior to this symbol. Therefore, a simple representation of the signature could be as follows: (*application, command, field.name, %n*). The accuracy of this signature can be verified by removing these symbols from a vaccine to test the vulnerable program. If the exception disappears with the occurrences of the symbol, we are assured that the signature carries no false positives.

Signatures using protocol information such as field length were first proposed by Liang et al. [Liang and Sekar 2005b]. However, their approach does not offer a means to estimate the size of a vulnerable buffer which is important for generating an accurate signature. A weakness of such signatures is that they may not work well on binary protocols with fixed field lengths. In this case, our BVI approach can still generate a token-sequence signature.

Packet vaccine is also able to test borderline values using protocol information. For example, a borderline value of the field “total questions” in a DNS packet is zero [Mockapetris 1987]. A vaccine can be generated by setting the field to zero. After injecting the vaccine into a vulnerable “named” server, nonexception indicates that the field value should be larger than zero.

3. EVALUATION

We evaluated packet vaccine using a proof-of-concept implementation. In this section, we first describe this implementation and then present our experimental results and analysis, which include vaccine effectiveness and signature quality.

Our experiments were carried out on two Linux workstations: one with Redhat 7.3 operating system, Intel Pentium 4 1.5GHz CPU and 256MB memory, and the other with Redhat 6.2, Pentium 3 1GHz CPU and 256MB memory. We used the Redhat 7.3 system for all experiments except those involving the Bind TSIG exploit, which requires Redhat 6.2.

We also used several network traces to evaluate the qualities of the signatures generated by our approach. Our dataset includes a trace of one million HTTP headers and one million DNS flows in and out of Indiana University.

²These bytes turned out to be the local variables lying between the function return address and the vulnerable buffer.

3.1 Implementation

We implemented packet vaccine on Linux. The target address set T is extracted from an application's process `proc` files, including `maps` and `status`, and sent to a vaccine generation module. This module scans the dataflow of a recorded session for the byte sequences inside T , scrambles their most significant bytes, creates a socket to convert the new dataflow into vaccine packets and transports them to the application. On the systems running the application, we installed a process monitor developed using `ptrace`, which serves as a collector to gather the contents of important registers should an exception happen to the process being monitored. Registers important to vulnerability diagnosis are CR2 and EIP. However, CR2 can only be accessed in the kernel mode. In our research, we developed a kernel patch for Linux 2.4.18 to read its content.

The signature generation engine has two components, a *prober* and a *verifier*. The prober tests an application using vaccines to identify signature tokens. It can work remotely. The verifier monitors processes for exception signals, and restarts the application if it is single-processed. In our implementation, the verifier was embedded in the `ptrace`-based monitor. On starting signature generation, the prober first makes a persistent connection with the verifier, and then sends a vaccine packet to the application. If the application's process crashes, the verifier intercepts the exception signal and notifies the prober through the connection. Otherwise, the verifier waits for a period of time (longer than the maximum crash time) before signaling that no exception has occurred. Our implementation supports both the BVI and AVI algorithms and can generate token-sequence and application-level signatures. A user can set either signature generation model by modifying a configuration file. For simplicity, we only implemented the sequential vaccine injection in our prototype system, which unfortunately introduced some performance penalties. In our experiments, we found that some applications could take tens of milliseconds to crash (Table I). The gross delay caused by crashes of multiple processes could be greatly reduced by a parallel approach.

3.2 Vaccine Effectiveness

A paramount question for packet vaccine is a vaccine's ability to detect an exploit. We address this question through an empirical evaluation reported in this section. We carried out experiments on real exploits of seven vulnerable applications obtained from SecurityFocus [SecurityFocus 2006]. They have been widely used for evaluating other techniques. In our research, we made sure that all these exploits were successful in the vulnerable applications by spawning a remote shell before testing them with our technique.

Packet vaccine successfully detected these exploits, and additionally diagnosed the related vulnerabilities to generate precise signatures. The details of exploits and detection results are listed in Table I. Note that due to time and resource constraints, we only implemented our prototype on Linux. For this reason, our experiments did not include exploits of Windows applications. However, we analyzed another 19 exploits which include Windows-based

Table I. Exploit Detection

Exploits	BID	Vulnerability Type	Exploit Packet Len	Detected	Number of Address-like Tokens	Max. delay for a crash
BIND tsign	2302	stack-based buffer overflow	510	Yes	3	0.0242s
Light httpd	6162	stack-based buffer overflow	231	Yes	13	0.0660s
ATP httpd	8709	stack-based buffer overflow	820	Yes	90	0.0265s
Samba	7294	stack-based buffer overflow	3097	Yes	26	0.0231s
OpenSSL v2	5363	heap-based buffer overflow	474	Yes	4	0.0768s
wu-ftpd	1387	format string attack	435	Yes	1	0.0732s
rpc.statd	1480	format string attack	1076	Yes	8	0.0043s

exploits such as Code Red II. We found none of their syntax and semantics would be damaged by our approach. This implies that packet vaccine can also detect them. Our analysis is presented in Appendix B.

We installed all vulnerable applications, except Bind, on the host running Redhat 7.3. Our implementation automatically extracted these applications' stack and heap information and estimated their address ranges using $\alpha = 2$. We present the details in Appendix C. ASCII-dominant protocols, such as HTTP and FTP, do not have suspicious jump addresses in their syntax strings and valid parameters, though these tokens may appear on packet payloads of these protocols (e.g., a binary file downloaded from a Web site). Therefore, vaccines seem very effective for detecting exploits using these protocols. In our experiments on light httpd, ATP httpd and wu-ftpd, our implementation blindly randomized all 4-byte sequences on exploit packets falling in these applications' target address sets, and caused SIGSEGV exceptions to their processes. We spotted randomized tokens in register CR2, which allowed us to correlate the exceptions to their causal packets.

Binary protocols could have a close-to-even distribution of byte sequences, rendering address-like sequences more likely to appear. However, those sequences may not coincide with the tokens necessary for an exploit, which usually constitute a small set (see the signatures in Table II). In our research, we put our approach to the test against exploits of four applications using binary protocols, including Bind-TSIG, samba, rpc-statd and openssl. Vaccines generated from these exploits all caused SIGSEGV. Again, the contents of CR2 confirmed correlations. For the stack-based buffer overflow (samba) and the format-string exploit (rpc-statd), the randomized tokens were found in CR2. An exception is Bind-TSIG, we found the token in EIP, as our kernel patch did not work for Redhat 6.2 and therefore we could not access CR2 under that Linux version.

Detecting a heap-based overflow turned out to be a little trickier. In the experiment on openssl, the value of the byte sequence we got from CR2 was larger than that of the randomized token by 12. We explain this as follows. The

Table II. Signatures and Their Generation Time. A Token in a Byte Sequence Signature is Represented as $i - j(B_i, \dots, B_j)$ ($i \leq j$), where i and j are the Positions of the Individual Bytes on the Token and B_i is a Byte's Hexadecimal Value. For Example, 229-230(0a,0a) Indicates that the Token 0x0a0x0a Lies Between the 229th and the 230th Bytes in the Payload. The Position Information is *optional* and not Useful for Text-Based Protocols Such as HTTP

Exploits	App. Signature	Gen. Time	Byte Sequence Signature (Hex)	Gen. Time
BIND tsig	—	—	4-12(00,01,00,00,00,00,01,3c), 73(3e),134(0c),147(31),197(0c),210(3e), 273(3e), 336(1e),367(10),384(3e), 447(34),500(00),505-507(00,00,fa)	4.881s
Light httpd	(., 'GET', filename, 178)	0.345s	0-3(47,45,54,20), 229-230(0a,0a)	1.360s
ATP httpd	(., 'GET', filename, 703)	0.274s	0-4(47,45,54,20,2f), 818(0a)	2.708s
Samba	(., 'TRANS2_OPEN2', filename, 2000)	0.622s	0-2(00,04,08),4-8(ff,53,4d,42,32), 28-29(01, 00),32-33(64,00),37-40 (d0,07,0c,00),55-56(d0,07),58-60 (00,0c,00),63-66(01,00,00,00)	7.636s
OpenSSL v2	(., 'Master Key', arguments, 298)	0.358s	0-11(81,d8,02,01,00,80,00,00,00, 80,01,4e)	5.012s
wu-ftpd	(., 'SITE', 'EXEC', %n)	0.130s	0-9(53,49,54,45,20,45,58,45,43,20), 431-432(25,6e)	4.228s
rpc.statd	(., 'STAT', name, %n)	0.116s	4-31(00,00,00,00,00,00,00,02,00,01,86, b8,00, 00,00,01,00,00,00,01,00,00,00, 01,00,00,00,20), 36-39(00,00,00,00,09), 60-63(00,00,00,00),68-74(00,00,00,00, 00,00,03),164-165(25,6e)	BVI-1: 5.780s
			4-31(00,00,00,00,00,00,00,02,00,01,86, b8,00, 00,00,01,00,00,00,01,00,00,00, 01,00,00,00,20), 36-39(00,00,00,00,09), 60-63(00,00,00,00), 68-74(00,00,00,00, 00,00,03),144-145(25,6e), 146-147(25, 6e),153-154(25,6e),160-161(25,6e), 162-163(25,6e),164-165(25,6e)	BVI-2: 19.717s

exploit took advantage of the “free()” function to overwrite a function's return address. The location of that address was faked as the content of a linking pointer in a bogus idle memory segment's heap management data structure. On the exploit's payload, the address of that segment's header was provided. That address was supposed to be lower than the linking pointer's address by 12. The exception happened when the heap management system attempted to access that linking pointer using the header's address which was randomized by our approach.

In addition, there are disjunctive tokens “%n” in the exploit packet of rpc-statd. BVI-1 captured the last one, and BVI-2 identified all of them, which are presented in Table II: 144-145 (25,6e), 146-147 (25,6e), 153-154 (25,6e), 160-161 (25,6e), 162-163 (25,6e), 164-165 (25,6e).

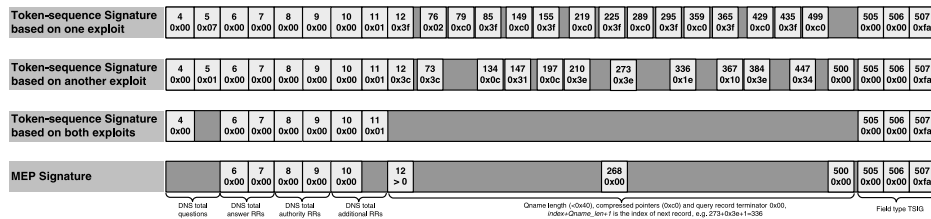


Fig. 4. Signatures for bind TSIG.

3.3 Signature Quality and Performance

A summary of results of our experiments on signature generation can be found in Table II. To evaluate the quality of our signatures, we compared them with signatures reported in recent literature [Brumley et al. 2006]. We report other results of our experiments on signature generation in Table II. A vulnerability-based signature can prevent all possible exploits on a vulnerability [Crandall et al. 2005]. Recently, Brumley et al. have proposed a gray-box approach to generate such a signature on the basis of *static analysis* of a vulnerable program’s binary code [Brumley et al. 2006]. Their technique intensively utilizes application information and theoretically promises to produce a perfect signature.

Brumley et al. described in their paper two *monomorphic-execution-path* (MEP) signatures, one for Bind TSIG and the other for ATP httpd. MEP signatures computed from a single exploit path are usually exploit-specific. Nevertheless, with the information extracted from the vulnerable application, they are still very accurate. Here, we analyze our signatures using these signatures.

3.3.1 Quality of the Token-Sequence Signature: Bind-TSIG. Bind is a very popular DNS server. It supports a secret-key transaction authentication in which messages bear transaction signatures (TSIG). There is a buffer-overflow vulnerability in Bind 8.2.x which allows an attacker to gain control of a system running Bind. This vulnerability can be exploited through both UDP and TCP queries. Our experiments were on UDP-based exploits and Bind 8.2.2.

We ran the BVI algorithm against two exploits of the Bind-TSIG vulnerability and generated signatures for them. The packets of these exploits share 19 bytes at the same locations on their application payloads. Based on these bytes, the BVI algorithm generated another signature with 10 bytes. These signatures³ are presented in Figure 4, along with the MEP signature.

All signatures include bytes 6 to 10 which are zero and bytes 505 to 507 which are 0x0000fa (a zero-length Qname followed by the field type TSIG). These bytes are indispensable to a successful exploit, as we discovered from Bind’s

³Our signature may also include the target address set, which we believe does not make the signature too specific for a control-flow hijacking attack. This is because that set includes all possible jump targets, not a specific address.

source code. Besides them, our signatures also contain some other tokens which are described below.

Bytes 4 to 5 are the number of queries inside the packet. Byte 4 must be zero for the UDP-base exploit due to the size limit of a UDP-based packet. However, byte 5's content is overly specific in our first two signatures because the number of queries is seven in the first exploit and one in the second one. On the other hand, that byte must be nonzero, which has not been captured by the MEP signature. Similarly, byte 11, along with byte 10, is the "ARcount" field which indicates the number of resource records in the additional records part. It must be nonzero to accommodate the TSIG field, but our signatures are overly specific in determining its value. Byte 12 appears on both our first two signatures and the MEP signature, but ours specify its content. Bytes between 73 and 500 on our first two signatures are also specific. These bytes serve as the length octets in the "Qname" field of a query, which are important for the successful parsing of a DNS query. However, an attacker may change the structure of the exploit packet to avoid these bytes. This problem is hard to avoid with only a single instance of the exploit and no application information at all. It can be mitigated if the signature is constructed from more than one exploit instance, as the third signature in the figure.

The MEP signature also has some problems. It misses bytes 4, 5, and 11, and also contains overly specific tokens, such as bytes 268 and 500. Byte 500 is also present in our signature from the second exploit while not in the first exploit. Both bytes (i.e., 268 and 500) signal the end of a query in a particular exploit. However, the attacker can avoid them by changing an exploit packet's structure, such as the number of questions and their sizes. For example, byte 268 has a nonzero value in two exploits used in our research.

Using the block-searching technique, a sequential BVI algorithm took 5.732 and 4.881 seconds to generate the first two signatures. We believe an optimized implementation and introduction of parallelization could improve that performance. The third signature was generated within 0.2 seconds. The generation time for the MEP signature has not been given by the authors of that technique [Brumley et al. 2006].

3.3.2 Quality of the Application-Level Signature: ATP-httpd. We also compared our application-level signature for ATP-httpd with the MEP signature in [Brumley et al. 2006]. ATP-httpd contains a vulnerable buffer which will be overrun by a requested filename longer than 680 bytes. Built upon the analysis of the program's binary code, the MEP signature contains richer information than ours. It points out the HTTP command which leads to the vulnerability could be either "GET" or "HEAD", while our signature only identifies "GET" from a single exploit instance. However, the MEP signature contains two overly specific tokens – "/" and "/" which actually are parts of the shell code (Figure 5). We scrambled both tokens in the exploit payload to generate a vaccine, which did not prevent the vaccine from triggering a SIGSEGV exception. In addition, the field length identified by their signature is 812 bytes, which is not necessary for an exploit. Our signature offers a better estimate of the vulnerable buffer size. The AVI algorithm determined the maximal length of

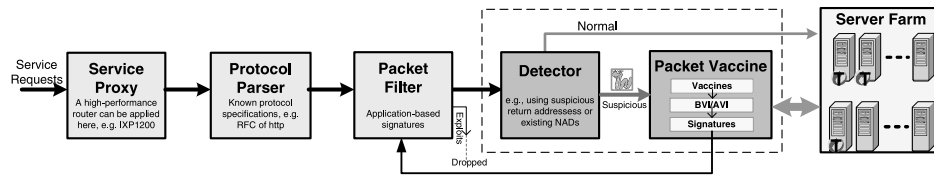


Fig. 6. An architecture to protect Internet servers using packet vaccine. Label *T* indicates test servers.

servers, while the HTTP traces were collected from edge routers, containing the traffic for other HTTP servers such as Apache or IIS which accommodate a longer filename field than ATP-httpd and light-httpd could. For example, Apache uses a dynamically allocated buffer to host HTTP requests and therefore can handle a request with the filename field longer than 178 bytes; however, the same request will cause a buffer-overflow in light-httpd.

4. EXAMPLE APPLICATION: PROTECTING INTERNET SERVERS

In the section, we present an architecture which applies packet vaccine to protect Internet servers from remote control-flow hijacking attacks. This architecture serves as an example to demonstrate the potential application of our technique. Other applications are discussed in Section 5.

We prototyped the architecture under Linux to protect Apache HTTP servers. In this section, we also include a performance study of this proof-of-concept system.

4.1 Architecture

Figure 6 illustrates the architecture we propose. A service request is first intercepted and cached by a service proxy and parsed by a parser. The parser is optional here and only useful when we use application-level signatures. Then, the request is screened by a filter which identifies and drops known exploits using exploit signatures. Behind the filter, a *detector* examines the request and labels it as either *normal* or *suspicious*. The detector could simply be part of our packet vaccine mechanism, which classifies packets with regards to the appearance of address-like tokens on their payloads. Alternatively, we could take other simple detection techniques, such as the one which identifies the packets with overlong fields. After classification, a *normal* request is forwarded to a server farm directly, while a *suspicious* request triggers the packet vaccine mechanism which acts as discussed in Section 2. If that request does contain an exploit, packet vaccine generates a new signature and adds it to the filter. Otherwise, the proxy forwards the original request to the server farm.

The architecture makes use of a small set of *test servers*⁵ in the server farm to test vaccine packets. A test server has a collector on it, which serves to

⁵Separate test servers are not indispensable in our architecture. A server providing live services can also be used to test vaccines (e.g., through forking a test process). Similar techniques are adopted in Shadow HoneyPot [Anagnostakis et al. 2005], Sweeper [Tucek et al. 2007], and FLIPS [Locasto et al. 2005].

glean registers' contents should an exception happen. Signature generation can also happen on such a server. Our current implementation of the architecture only works for stateless services such as HTTP. To accommodate stateful services such as FTP, the test server also needs a checkpoint/rollback (CR) mechanism to recover the state before each test, which unfortunately incurs overhead. Advances in CR techniques offer the opportunity to greatly reduce such overhead: for example, FlashBack [Srinivasan et al. 2004; Tucek et al. 2007] makes a checkpoint for a process by simply forking a backup process, which enables a very efficient recovery once the original process is crashed by an exception. Application of such techniques to our architecture is left as future research.

A threat to this architecture is denial-of-service (DoS) attacks: an attacker can generate a large number of packets containing address-like tokens, in hopes of exhausting the capacity of the test servers. However, this will not affect legitimate client requests if their packets do not carry such tokens, as these packets will not go through the test servers. The attacker could also intentionally crash a vulnerable program by exploiting its vulnerability by using jump addresses outside the program's address space. Such an attack cannot be detected by the current design of packet vaccine, as the exploit packet may not include any address-like tokens. Defenses against this threat would presumably need to generate a signature from features of packets other than the presence of address-like tokens, such as an overly long application field. Such approaches are left for future study.

4.2 Performance Study

To implement a prototype system for HTTP service, we developed a service proxy and a filter (including an HTTP parser), and combined them with our implementation of packet vaccine (Section 3.1) which contains a detector. A suspicious client request is checked by the exploit detection module before being forwarded to real servers, whereas the reply of that request is directly sent back to the client. The proxy caches a request first and then handles it according to the detection result, either dropping it as a convicted exploit attempt or forwarding it to an HTTP server as an innocent request. Our filter simply examines packet fields output from the parser and drops the request whose characteristics match to a known exploit signature. Since HTTP is a stateless service, we did not implement the process-level CR in this prototype. However, an extension of our implementation can easily accommodate a stateful service, as CR mechanisms [Dunlap et al. 2002; Srinivasan et al. 2004] can be directly integrated to our system.

Over the prototype system, we carried out a performance test. Two hosts were used in our experiment, one for both the proxy and the test server and the other for the web server. Both were equipped with 2.53GHz Intel Pentium 4 Processor and 1 GB RAM, and running Redhat Enterprise 2.6.9-22.0.1.EL. They were interconnected through a 100MB switch. We utilized an Apache 2.0.55 to provide web service. In our experiment, we evaluated the performance of our implementation from the following perspectives: (1) *Server*

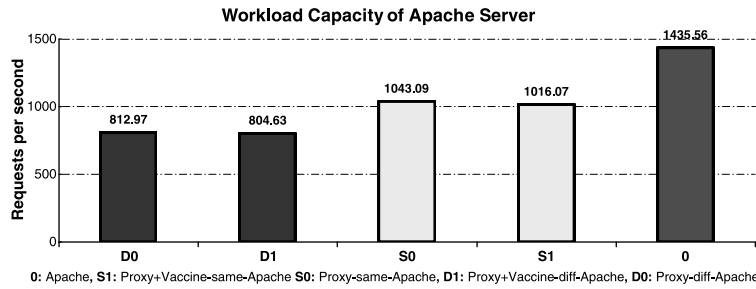


Fig. 7. The workload capacities in five different server settings.

overheads, where we compared the workload capacity of our implementation with that of an unprotected Apache server; (2) *Client-side delay*, where we studied the average delay a client experiences under different test rates.

4.2.1 Server Overheads. We tested the workload capacity using ApacheBench (ab) 2.0.41-dev which comes bundled with the Apache source distribution. ApacheBench is a tool for benchmarking the Apache Web server. In our experiment, we measured the workload capability in terms of requests processed per second (requests/second) under the following five server configurations: (0) “Apache only” (*Apache*), (D0) “Apache and the proxy on different hosts” (*Proxy-diff-Apache*), (S0) “Apache and the proxy on the same host” (*Proxy-same-Apache*), (D1) “Apache and the packet-vaccine proxy (with the parser and the detector) on different hosts” (*Proxy+vaccine-diff-Apache*), (S1) “Apache and the packet-vaccine proxy on the same host” (*Proxy+vaccine-same-Apache*). During the experiments, we invoked ApacheBench for each server configuration as follows: “[root@localhost ~]# ab -n 100000 -c 100 http://192.168.1.13/test.html”. In response to this command, ab generated 100000 requests in total to the Web server with the IP address 192.168.1.13 for downloading the Web page test.html, maintaining 100 requests outstanding concurrently. The size of the Web page is 7.5KB. After all these requests were served, ab output the average number of requests processed by the server in a second.

Figure 7 illustrates the experiment results. At a first glance, it seems that our implementation brought down the Apache’s performance by about 44% in the setting *Proxy+vaccine-diff-Apache* and about 29% in the setting *Proxy+vaccine-same-Apache*, which is quite unpleasant. A close look at the results, however, reveals that the major performance penalty came from the proxy. The homegrown proxy used in our proof-of-concept implementation could not catch up with the high-performance Apache and therefore dragged down the performance of the whole system. Simply adding the proxy into the system introduced about 43% performance penalty in *Proxy-diff-Apache* and 27% in *Proxy-same-Apache*. On the other hand, the packet vaccine components worked pretty fast. They affected the performance only by one to two

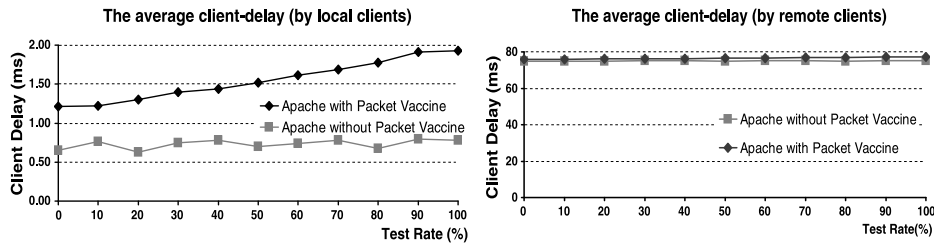


Fig. 8. The average delay experienced by a local or remote client.

percent. Therefore, we tend to believe that a high-performance HTTP proxy could greatly improve the workload capability.

Actually, an application-level proxy is not required for the architecture. An alternative could be a firewall or a load balancer built upon IXP network processor, as used in [Anagnostakis et al. 2005]. This could remove the role the proxy played as a performance bottleneck. Another observation in our experiment is moving the proxy to the host running Apache improves the workload capability by at least 10%. Therefore, it seems that making the proxy an Apache module might greatly reduce the server overhead.

4.2.2 Client-side delay. Once the detector identifies a suspicious request, a round of exploit detection will be triggered to test that request. This introduces delay to a legitimate client if the request turns out to be innocent. Here, we call the ratio of the service requests being tested the *test rate*. If the test rate increases, the average delay experienced by a legitimate client will also increase. In our experiment, we studied the change of the client-side delay against different test rates. We carried out both a local experiment within Indiana University’s campus network and a cross-campus experiment between Indiana University and North Carolina State University. The experimental results are presented in Figure 8.

As we expected, the average delay for a local client increased almost linearly with the test rate. However, this result could be misleading, as the local client experienced much smaller round trip delay (RTD) than an average Internet user: the RTD in a campus we measured is around $300\mu s$, while the average RTD on the Internet is around $100ms$. Therefore, an Internet client’s perception of the presence of packet vaccine could be completely overshadowed by the RTD. This was confirmed in the cross-campus experiment: as presented in Figure 8, the $75ms$ RTD between the two campuses dominated the client-side delay, making the $1ms$ overhead of our protection mechanism negligible.

In summary, packet vaccine does introduce performance penalty to the server but such penalty is acceptable if weighed against the security enhancement it offers. On the other hand, the client-side overhead is almost negligible, completely dwarfed by the RTD an average Internet client experiences.

5. DISCUSSION

5.1 Other Applications

Network-based honeypots and signature generators [Kreibich and Crowcroft 2004; Yegneswaran et al. 2005; Singh et al. 2004] usually detect new exploits and generate their signatures earlier than a host-based system. However, these techniques can suffer from a high false positive rate due to the interference of Internet noise or even poisoning attacks [Richardson et al. 2005; Perdisci et al. 2006]. Therefore, it is important to use the vulnerable program to validate the exploit detected by these approaches. This is a daunting task because there could be hundreds of different implementations of the protocol, plus various versions and patching levels for individual implementations.

Our packet vaccine mechanism offers a simple solution to this problem. A network-based detector can forward the dataflow recorded from a suspicious communication session to the administrator of a local-area network. Using our technique, the administrator can generate vaccines from such dataflow to scan the systems inside his administration domain. If any system reports an exception, not only do we confirm the existence of the exploit but we can generate an accurate signature. Otherwise, the administrator can conclude that either the dataflow does not reflect any real exploit, or no program in his administrative domain contains the related vulnerability. Similarly, packet vaccine can be used as a “vulnerability assessment tool.” We can probe servers, switches, routers, and workstations for conditions that, if left untreated, can result in penetrations.

5.2 Limitations

Packet vaccine may have false negatives in exploit detection, since there is a possibility that our approach modifies an exploit’s semantic tokens. This is more likely to happen for the applications using binary protocols, though so far we have not found an example “in the wild.” Several ways of mitigating this problem have been discussed in Section 3.2. A simple yet effective solution is generating multiple vaccines, each randomizing one byte on an address-like token. In this way, if the exploit’s semantic tokens survive any of these randomization, our approach will detect the exploit.

Our approach requires decoding a packet before generating a vaccine. It is nontrivial to support a variety of encoding schemes while maintaining system performance. This problem could limit the potential for using our technique at the edge of a network to protect a wide range of Internet services and applications. We plan to seek a good solution to the problem in the future research. Our current design, however, is just meant to protect a specific application, and therefore can focus on the encoding schemes used in that application. For example, a vaccine mechanism for protecting an Apache server only needs to know how to decode the encoding schemes that Apache supports.

Our approach will not work directly on packets with encrypted payload or checksums. In this case, we need an application-level proxy to decode these packets and construct new packets for vaccine generation [Wang et al. 2006;

Locasto et al. 2005]. Another approach is to interpose on the API calls to the functions for decryption so as to extract and examine the plaintext before it being processed by an application. This can be achieved using API interception tools such as Detours.⁶

Both types of signatures we used in our research are limited in their capabilities to represent necessary exploit conditions. For example, `null-httpd` contains a vulnerability which allows one to specify a smaller buffer while supplying a longer payload. An ideal signature is to check whether the real payload size matches the specified size. However, none of our signatures can describe this condition. We leave it to future work to examine how to use our black-box techniques to acquire information for more expressive signatures [Wang et al. 2004; Brumley et al. 2006].

6. RELATED WORK

In this section, we survey the previous work related to packet vaccine. Research on exploit detection and automatic signature generation is most relevant to our technique. Existing approaches in this area can be loosely classified into three categories, that is, network-based approaches, host-based approaches and hybrid approaches. Another related area is software robustness testing. Below we compare the work in these areas with our approach.

6.1 Network-Based Approaches

Network anomaly detection has been widely used to detect exploit attempts from network traffic. Scan detection [Telescope 2006; HoneyNet 2006; Spitzner 2003] captures anomalous scanning activities which usually herald massive spreads of Internet worms. More active honeypots such as HoneyNets can cheat attackers into revealing the packets carrying exploit payloads [Crandall et al. 2005; Spitzner 2003; HoneyNet 2006]. Recently, research on hitlist worms has also received much attention [Shannon and Moore 2004; Zou et al. 2005; Whyte et al. 2005].

Many NADs look for anomalous patterns in packets to discover attack dataflow [Yegneswaran et al. 2005; Wang and Stolfo 2004; Toth and Krügel 2002; Kruegel et al. 2005]. Examples include Nemean [Yegneswaran et al. 2005] which builds vulnerability-specific signatures for unknown exploits using protocol specifications, PayL [Wang and Stolfo 2004] which senses the large deviation in the byte frequency distribution on anomalous packets, Toth and Kruegel's scheme [Toth and Krügel 2002] which builds upon abstract execution of attack payloads. Structural information of executables within an exploit is also proposed to serve as a fingerprint for possible exploits, though detection of such information is very computation-intensive [Kruegel et al. 2005], and is prone to false positives (FPs), as legitimate packets might also contain executables, such as OS updates and patches.

Using the attack dataflow detected, network signature generators automatically create attack signatures. Earlybird [Singh et al. 2004], Honeycomb

⁶<http://research.microsoft.com/sn/detours/>

[Kreibich and Crowcroft 2004], Autograph [Kim and Karp 2004] and SweetBait [Portokalidis and Bos 2005] use a prevalent substring across worm-related packets as a signature. This treatment could be defeated by polymorphic worms capable of altering their forms in every new infection. Polygraph [Newsome et al. 2005] proposes to detect polymorphic worms using multiple short invariants which are hard to evade even for polymorphic worms. PADS [Tang and Chen 2005] identifies anomalous packets using a position-aware distribution of bytes.

Anomaly detection solely relying on network information is usually less accurate than host-based approaches, tending to produce FPs. On the other hand, these approaches are usually more efficient: some of them can even work at the link speed. Our approach can also serve to improve the accuracy of the signatures generated by these approaches. A vaccine can be made through randomizing the address-like tokens on the content of such a signature, which is usually composed of some prevalent substrings in anomalous dataflow. If the original signature indeed relates to an exploit, this vaccine could cause an exception to a vulnerable program.

6.2 Host-based Approaches

Host-based approaches make use of host information to detect anomalies and generate signatures. As exploits actually happen on a host, these approaches are usually more accurate than network-based approaches, incurring fewer FPs. However, they tend to be much slower. TaintCheck [Newsome and Song 2005], VSEF [Newsome et al. 2005], Sweeper [Tucek et al. 2007], Minos [Crandall and Chong 2004], and Vigilante [Costa et al. 2005] track the dataflow from the network to the point where an anomaly happens, for example, jumping to an address offered by the input data. DACODA [Crandall et al. 2005] can monitor the execution flow of the whole system, including the anomaly happening across multiple processes. A major problem for these approaches is performance which typically drops by at least one order of magnitude. Suh et al. [2004] proposes an approach similar to the NX technique [Wasson 2004] to improve the performance of dynamic information tracking, which however needs a hardware support. In contrast, our vaccine mechanism tracks suspicious dataflow in a black-box fashion, which is significantly faster than these gray-box approaches and still preserves most of their accuracy and usability.

Recently, Liang et al. and Xu et al., independently propose two approaches [Xu et al. 2005; Liang and Sekar 2005b] which use memory address-space randomization (ASR) to foil exploit attempts, and then automatically generate signatures through forensic analysis of the related exceptions. These approaches attempt to fix the de-randomization weakness of ASR discovered by Shacham et al. [2004]. In particular, COVERS [Liang and Sekar 2005b], proposed by Liang et al., first proposes a novel construction of application-level signature which uses field length to characterize a buffer overflow vulnerability. Although we also use this signature, our AVI technique augments their approach by making an accurate estimate of the field length. This could prevent a derandomization attacker from repeatedly probing a vulnerable program by

shrinking the field length of the attack packet byte by byte. In addition, our technique also offers a more reliable way to correlate exceptions with the exploit packets, on which we elaborate in Section 2.2.

6.3 Hybrid Approaches

Hybrid approaches combine network-based and host-based approaches together to seek a trade-off between performance and accuracy. Bro +Apache [Dreger et al. 2005] and ARBOR [Liang et al. 2005; Liang and Sekar 2005a] utilize host-based and network-based context information in parallel to improve detection accuracy. Shadow Honeypot [Anagnostakis et al. 2005] employs these two techniques in a sequential way, using an NAD to detect anomalous packets and a honeypot to analyze them. FLIPS [Locasto et al. 2005] and the software self-healing approach proposed in Locasto et al. [2006] use a similar architecture to protect Internet services, relying on STEM [Sidirolou et al. 2005] to detect exploit attempts.

Shadow Honeypot, FLIPS and the approach in [Locasto et al. 2006] all rely on code instrumentation of a service program to detect exploit attempts, and therefore require the knowledge of source code or recompiled binaries. This makes these approaches hard to apply to protect commodity software. By comparison, packet vaccine works well over commodity software.

HACQIT [Reynolds et al. 2003] invokes a test process after an exploit crashes a protected program, and replays suspicious packets to a sandbox running the same program to monitor whether the same exception happens again. However, this approach only works for denial-of-service attacks and does not offer a reliable means to establish a correlation between the exception and the exploit inputs.

6.4 Software Robustness Testing

The vaccine-injection technique can trace its root to software robustness testing, especially software-implemented fault injection (SWIFI) [Musa et al. 1996]. Software testing is a process to evaluate the quality of developed software, which includes its correctness and security property. Fault injection is a software testing and evaluation method which involves inserting faults into a system to determine its response to these faults. Some early methods to automatically test operating systems for robustness include the Crashme program [Carrette 2006], the Fuzz project [Carrette 2006], the FIAT system [Barton et al. 1990], the FERRARI system [Kanawati et al. 1995] and the FTAPE system [Tsai and Iyer 1995]. A recent work in this area is CMU's Ballista [Ballista 2006], which automatically probes an operating system's APIs with a large number of valid and invalid inputs to evaluate their robustness.

Similar testing techniques can also be applied to assess the effectiveness of a security mechanism. Vigna et al. present a testing tool which automatically generates variations of a known exploit to evaluate an intrusion detector's accuracy [Vigna et al. 2004]. A similar approach is the Thor tool [Marty 2002], which tests intrusion detection systems using variations of known attacks.

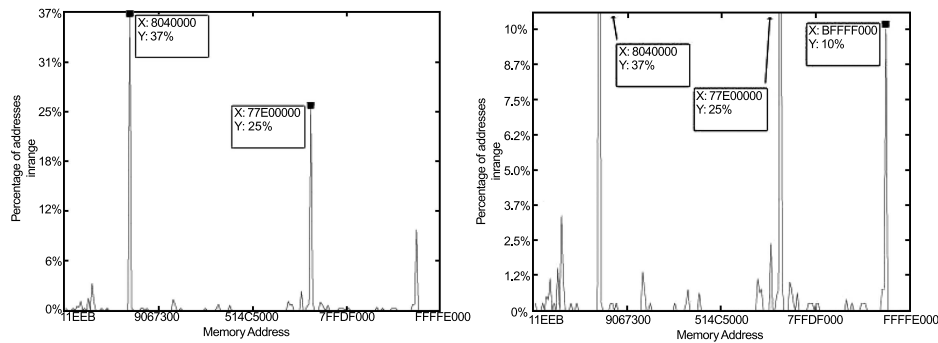


Fig. 9. Distribution of jump addresses (counting duplicated addresses).

Software robustness testing techniques have paved the road to building vaccine-based defense. However, our approach differs fundamentally from these approaches in its ability to quickly generate protective signatures to “heal” the vulnerabilities being discovered. Another important difference is that our approach does not randomly generate faults as many fault-injection tools do. Instead, packet vaccine takes advantage of anomalous packets or confirmed exploits to direct vaccine generation, which significantly increases the chance to identify an unknown vulnerability.

7. CONCLUSIONS AND FUTURE WORK

In this article, we presented packet vaccine, a fast and black-box technique for exploit detection, vulnerability diagnosis and signature generation. We described its design and provide examples for its application. We also implemented a proof-of-concept prototype, and evaluated our technique using it. Our experimental results demonstrate the effectiveness of our technique, which successfully captures real exploits and generates accurate signatures, and its efficiency, which greatly improves over the gray-box approaches and works well in protecting online services.

APPENDICES

A. Distribution of Exploitable Jump Addresses

We inspected the jump addresses of all exploits recorded in MilkWorm⁷, which includes numerous attacks on Windows, Linux, Solaris, and BSD operating systems. Figures 9 and 10 illustrate the distribution of these addresses. (Note that the figure on the right offers a closer look at the figure on the left; see the difference on their Y-axes.) The distribution illustrated in Figure 9 counts in duplicated addresses, while the distribution in Figure 10 does not. In other words, if an address appears twice in our dataset, we counted it twice in Figure 9 and once in Figure 10. From these figures we observed a remarkable

⁷<http://www.milkworm.com>

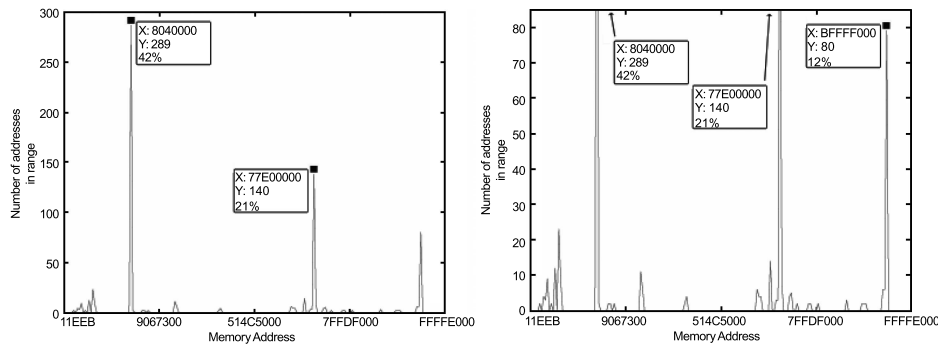


Fig. 10. Distribution of jump addresses (not counting duplicated addresses).

concentration of jump addresses. The spike around `0x08040000` includes the memory addresses used by the heap, GOT and BSS-based overflow exploits. Around `0xbffff000` are the addresses in stack-based overflow exploits. Attacks on Windows applications intensively use the addresses around `0x77e00000`, as illustrated in the figures.

It is quite clear that most of these addresses concentrate on very narrow ranges. For example, the top three spikes cover 72% of exploits. This suggests that we could detect most attacks by monitoring a small memory address range. This does not mean that packet vaccine cannot protect the memory ranges around other smaller spikes. It is simply less efficient to do so, as it will result in more vaccines being generated and tested relative to the number of exploits that might be detected.

B. Exploit Analysis

In addition to the exploits reported in Section 3.2, we also analyzed the code of nineteen other exploits to evaluate the effectiveness of the vaccine. To limit the bias, we inspected all remotely-exploitable vulnerabilities on the first 16 pages of the securityfocus records [Vulnerabilities 2006] between 5/4/2006 and 4/17/2006, and all remotely-exploitable vulnerabilities with BugTrap IDs between 5000 and 5200. The results are listed in Table IV, in which “[]” stands for a blankspace, “[x]” for a variable-length string, “x . x” for a constant-length string and “JAJA” for a jump address.

Comparing the above syntax and semantic tokens with the distributions illustrated in Figure 9, we found that except the jump addresses, none of them fall into the address ranges intensively used by exploits. This suggests that scrambling address-like byte sequences on these exploits’ payload will not temper with any of these tokens. Therefore, we believe that packet vaccine can detect all these exploits.

C. Memory Ranges Used in our Experiments

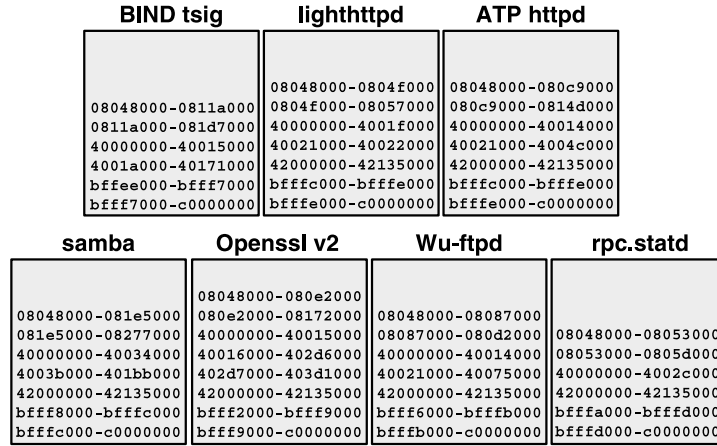
We list the memory ranges utilized in our experiments on vaccine effectiveness. All together, the heap and stack occupied less than 0.1% of the whole process virtual memory (i.e., 4MB/4GB) as we observed in the experiments. Though

Table IV. Exploit Packet Information. ¹Code Red II does not appear in Securityfocus, but we also analyzed it for illustration purposes

App Name	BID	Exploit Packet Format	Protocol Syntax	Exploit Semantic Tokens
Code Red II ¹	—	GET[/[x].ida?[x]%uJA %uJA[x]=[x][HTTP/1.1] \r\n\r\n	GET, [], HTTP/1.1, \r\n	.ida?=?,%u, AJAJA
Null HTTPD	6255	POST[/][HTTP/1.0\r \nContent-Length:-800\r\n \r\n[x]AJAJAxxxx\r\n	POST, [], HTTP/ 1.0,\r\n, Content-Length,;,	-800, AJAJA
Sami FTP Server	16370	USER[[x]AJAJA]\r\n	USER,[],\r\n	AJAJA
Sendmail	17192	Subject:[x]AJAJA]\r\n	Subject. :[],\r\n	AJAJA
ArGoSoft Server FTP	12755	DELE[[x]AJAJA]\n	DELE, [], \n	AJAJA
MySQL	17780	\x6f\x01\x00\x00 \x03SELECT[[x]AJAJA[x]	\x6f\x01\x00\x00 \x03, [], SELECT	AJAJA
Xine	17769	xine[[x]AJAJA[x]%n[x].mp3	xine,[]	%n,AJAJA
BL4 SMTP Server	17714	MAIL[FROM:[x] AJAJA[x]\r\n	MAIL FROM, :, [], \r\n	AJAJA
Solaris x86 nlps server	2319	NLPS:002:002:AJAJA[x] yahoo...\n	NLPS, :,\n	002,yahoo..., AJAJA
Qualcomm qpopper	948	LIST[[]][x]AJAJA[x]\n	LIST, [],\n	1, AJAJA
Novell GroupWise Messenger	17503	GET[/][HTTP/1.1\r \nAccept-Language: [x]AJAJA[x]\r\n\r\n	GET,[],HTTP/ 1.1,\r\n,Accept- Language,;	/, AJAJA
Sybase EA Server	14287	GET[[x]login.jsp?[x] AJAJA[x][HTTP/1.1\r \nAccept:[x]\r\nUser- Agent:[x] \r\nHost:[x]:[x]\r \nConnection:Close\r\n\r\n”	GET,[],?,HTTP/ 1.1,Accept,User- Agent,Host, Connection, close,;,\r\n	login.jsp, AJAJA
Microsoft SQL Server	5004	GET[/Nwind/Template/ catalog.xml?contenttype= text[x] AJAJA[x][HTTP/ 1.1\r\n\r\n	GET,[],=?, HTTP/1.1, \r\n	/Nwind/ Template/ catalog.xml, contenttype, text,/, AJAJA
AnalogX SimpleServer: WWW	5006	[x]AJAJA[x]\r\n\r\n	—	\r\n\r\n, AJAJA
Netscape Composer	5010	<html><body> <font[face=[x]AJAJA[x]>[x] </body> </html>	<,</,>,html, body,font,face,=	AJAJA
Apache	5033	GET[/][HTTP/1.1\r \nHost:[x]\r\nX-xxxxxxx:[x] \r\nX-xxxx:[x]AJAJA[x]\r \nTransfer-Encoding: chunked \r\n\r\n5\r\nxxxxx \r\n-146\r\n	GET,[],/,HTTP/ 1.1,\r\n,Host,;, \r\n, Transfer- Encoding,	X-, chunked, 5,-146,AJAJA
AnalogX Proxy	5138	\x04\x01\x00\x19\x00 \x00\x00\x01[x]\x00[x]AJAJA \xEB\x22\x00	\x04\x01\x00 \x19\x00\x00 \x00\x01,\x00	\xEB\x22, AJAJA
Nullsoft Winamp	5170	OK\r\n\r\n9.99\r \n\r\n[x]AJAJA[x]\r\n	OK,\r\n	9.99,AJAJA
MyWebSer- ver	5184	GET[[x]AJAJA[x]	GET,[]	AJAJA

Table V. Memory Usages

Exploits	heap(kb)	stack(kb)	vmData(kb)	vmStk(kb)	vmLib(kb)
BIND tsg	840	36	756	36	1456
Light httpd	28	8	32	8	1364
ATP httpd	516	8	528	8	1488
Samba	1652	16	584	16	2980
OpenSSL v2	616	28	584	28	5264
wu-ftpd	252	20	300	20	1652
rpc.statd	44	12	40	12	1360

Fig. 11. Memory ranges (T) used in our experiments.

public libraries take a larger memory space, we do not need to monitor all their memory ranges because there are only a few function entries that attackers can use for launching return-to-libc attacks. Only these pointers need to be covered by the target address set.

In Table V, we present the estimates of stack and heap usages in our experiments. The first two columns of the table were obtained from `/proc/pid#/maps`, in which the heap size refers to the size of the memory between `0x08048000` and `0x40000000`, and the stack size is the size of the memory between `0xc0000000` and the start of function libraries on a process's memory map. *vmData*, *vmStk* and *vmLib* are from `/proc/pid#/status` which record a process's current usages of the heap, stack and libraries. From the table, we can see *vmStk* is also the stack usage displayed in the memory map. Although there is some discrepancy between the heap usage and *vmStk*, they are pretty close.

Using these memory usages, we estimated the address ranges for the vulnerable applications in our experiments, which are listed in Figure 11. For simplicity, we let the range cover all public libraries though this is unnecessary, as we explained before. We set $\alpha = 2$ to keep a safe margin for the estimates of the stack and heap ranges, using the stack and heap sizes recorded in *vmStk* and *vmData*. For example, we estimated the heap size of ATP httpd by doubling the value recorded in its *vmData* (528kb), which was achieved through adding a set

of addresses from 080c9000 to 0814d000; similarly, we also included bfffc000-bfffe000 to its stack.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their insightful comments.

REFERENCES

- ANAGNOSTAKIS, K. G., SIRIDOGLOU, S., AKRITIDIS, P., XINIDIS, K., MARKATOS, E., AND KEROMYTIS, A. 2005. Detecting targeted attacks using shadow honeypots. In *Proceedings of the USENIX Security Symposium (SECURITY'05)*.
- ASSOCIATED PRESS. 2006. Microsoft warns against outside fixes. http://biz.yahoo.com/ap/060331/microsoft_s_security_snags.html?v=4.
- BALLISTA. 2006. The Ballista@ Project: COTS Software Robustness Testing. <http://www.ece.cmu.edu/~koopman/ballista/>.
- BARTON, J. H., CZECK, E. W., SEGALL, Z. Z., AND SIEWIOREK, D. P. 1990. *Fault injection*.
- BRUMLEY, D., NEWSOME, J., SONG, D., WANG, H., AND JHA, S. 2006. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the IEEE Symposium on Security and Privacy (SSP'06)*.
- CARRETTE, G. J. 2006. CRASHME: Random input testing. <http://people.delphiforums.com/gjc/crashme.html>.
- COSTA, M., CROWCROFT, J., CASTRO, M., ROWSTRON, A. I. T., ZHOU, L., ZHANG, L., AND BARHAM, P. T. 2005. Vigilante: End-to-end containment of internet worms. In *Proceedings of 19th ACM Symposium on Operating Systems Principles (SOSP'05)*. 133–147.
- CRANDALL, J. R. AND CHONG, F. T. 2004. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-34)*. 221–232.
- CRANDALL, J. R., SU, Z., AND WU, S. F. 2005. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*. 235–248.
- CRANDALL, J. R., WU, S. F., AND CHONG, F. T. 2005. Experiences Using Minos as a Tool for Capturing and Analyzing Novel Worms for Unknown Vulnerabilities. In *Proceedings of the GI International Conference on Detection of Intrusions & Malware, and Vulnerability Assessment (DIMVA'05)*. 32–50.
- CUI, W., PEINADO, M., WANG, H. J., AND LOCASO, M. E. 2007. Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *Proceedings of the IEEE Symposium on Security and Privacy (SSP'07)*. 252–266.
- DREGER, H., KREIBICH, C., PAXSON, V., AND SOMMER, R. 2005. Enhancing the accuracy of network-based intrusion detection with host-based context. In *Proceedings of the GI International Conference on Detection of Intrusions & Malware, and Vulnerability Assessment (DIMVA'05)*. 206–221.
- DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. 2002. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'02)*.
- HONEYNET. 2006. <http://www.honeynet.org/>.
- KANAWATI, G. A., KANAWATI, N. A., AND ABRAHAM, J. A. 1995. FERRARI: A flexible software-based fault and error injection system. *IEEE Trans. Comput.* 44, 2, 248–260.
- KIM, H.-A. AND KARP, B. 2004. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of 13th USENIX Security Symposium (SECURITY'04)*. 271–286.
- KREIBICH, C. AND CROWCROFT, J. 2004. Honeycomb: Creating intrusion detection signatures using honeypots. *SIGCOMM Comput. Comm. Rev.* 34, 1, 51–56.

- KRUEGEL, C., KIRDA, E., MUTZ, D., ROBERTSON, W., AND VIGNA, G. 2005. Polymorphic worm detection using structural information of executables. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID'05)*. 207–226.
- LIANG, Z. AND SEKAR, R. 2005a. Automatic generation of buffer overflow attack signatures: An approach based on program behavior models. In *Proceedings of the Annual Computer Security Applications Conference (CSAC'05)*.
- LIANG, Z. AND SEKAR, R. 2005b. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*. 213–222.
- LIANG, Z., SEKAR, R., AND DUVARNEY, D. C. April, 2005. Automatic synthesis of filters to discard buffer overflow attacks: A step towards realizing self-healing systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX'05)*.
- LOCASTO, M. E., SIDIROGLOU, S., AND KEROMYTIS, A. D. 2006. Software self-healing using collaborative application communities. In *Proceedings of the 13th Annual Network and Distributed Systems Security Symposium (NDSS'06)*.
- LOCASTO, M. E., WANG, K., KEROMYTIS, A. D., AND STOLFO, S. J. 2005. Flips: Hybrid adaptive intrusion prevention. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID'05)*.
- MARTY, R. 2002. Thor: A tool to test intrusion detection systems by variations of attacks. Master thesis, ETH Zurich.
- MEMVIEW. 2006. <http://www2.biglobe.ne.jp/sota/memview-e.html>.
- MICROSOFT. 2007. Microsoft debugging tools: Overview. <http://www.microsoft.com/whdc/devtools/debugging/default.aspx>.
- MOCKAPETRIS, P. 1987. DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION. RFC 3425. <http://www.ietf.org/rfc/rfc1035.txt>.
- MUSA, J., FUOCO, G., IRVING, N., JUHLIN, B., AND KROPFL, D. 1996. *Handbook of Software Reliability Engineering*. McGraw-Hill, New York, 167–216.
- NAUMOVICH, G. AND MEMON, N. D. 2003. Preventing piracy, reverse engineering, and tampering. *IEEE Comput.* 36, 7, 64–71.
- NEWSOME, J., BRUMLEY, D., AND SONG, D. 2005. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *Proceedings of the 13th Annual Network and Distributed Systems Security Symposium (NDSS'05)*.
- NEWSOME, J., KARP, B., AND SONG, D. 2005. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of IEEE Symposium on Security and Privacy (SSP'05)*. 226–241.
- NEWSOME, J. AND SONG, D. 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS'05)*. San Diego, CA.
- PASUPULATI, A., COIT, J., LEVITT, K., WU, S., LI, S., KUO, R., AND FAN, K. 2004. Buttercup: On network-based detection of polymorphic buffer overflow vulnerabilities. In *Proceedings of the IEEE/IFIP Network Operation and Management Symposium (NOMS'04)*.
- PERDISCI, R., DAGON, D., LEE, W., FOGLA, P., AND SHARIF, M. 2006. Misleading worm signature generators using deliberate noise injection. In *Proceedings of the IEEE Symposium on Security and Privacy (SSP'06)*. 17–31.
- PORTOKALIDIS, G. AND BOS, H. 2005. SweetBait: Zero-hour worm detection and containment using honeypots. Tech. rep. IR-CS-015, Vrije Universiteit Amsterdam.
- REYNOLDS, J. C., JUST, J., CLOUGH, L., AND MAGLICH, R. 2003. On-line intrusion detection and attack prevention using diversity, generate-and-test, and generalization. In *Proceedings of the Annual Hawaii International Conference on System Sciences (HICSS'03)*. 335.2.
- RICHARDSON, D. W., GRIBBLE, S. D., AND LAZOWSKA, E. D. 2005. The limits of global scanning worm detectors in the presence of background noise. In *Proceedings of the 2005 ACM workshop on Rapid malware (WORM'05)*. ACM Press, 60–70.
- SECURITYFOCUS. 2006. <http://www.securityfocus.com>.

- SHACHAM, H., PAGE, M., PFAFF, B., GOH, E.-J., MODADUGU, N., AND BONEH, D. 2004. On the effectiveness of address-space randomization. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'04)*. 298–307.
- SHANNON, C. AND MOORE, D. 2004. The spread of the witty worm. *IEEE Secur. Privacy* 2, 4 (July/August), 46–50.
- SIDIROGLOU, S., LOCASIO, M. E., BOYD, S. W., AND KEROMYTIS, A. D. 2005. Building a reactive immune system for software services. In *Proceedings of the USENIX Annual Technical Conference (USENIX'05)*. 149–161.
- SINGH, S., ESTAN, C., VARGHESE, G., AND SAVAGE, S. 2004. Automated worm fingerprinting. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*. 45–60.
- SPITZNER, L. 2003. Honey pots: Catching the insider threat. In *Proceedings of the Annual Computer Security Applications Conference (CSAC'03)*. 170–181.
- SRINIVASAN, S. M., KANDULA, S., ANDREWS, C. R., AND ZHOU, Y. 2004. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of USENIX Annual Technical Conference, General Track (USENIX'04)*. 29–44.
- SUH, G. E., LEE, J. W., ZHANG, D., AND DEVADAS, S. 2004. Secure program execution via dynamic information flow tracking. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*. 85–96.
- TANG, Y. AND CHEN, S. 2005. Defending against internet worms: A signature-based approach. In *Proceedings of the Annual IEEE Conference on Computer Communications (INFOCOM'05)*. Miami, FL.
- TELESCOPE. 2006. <http://www.caida.org/analysis/security/telescope/>.
- TOTH, T. AND KRÜGEL, C. 2002. Accurate buffer overflow detection via abstract payload execution. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID'02)*. 274–291.
- TSAI, T. K. AND IYER, R. K. 1995. Measuring fault tolerance with the ftape fault injection tool. In *Proceedings of the 8th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (MMB'95)*. Springer-Verlag, 26–40.
- TUCEK, J., LU, S., HUANG, C., XANTHOS, S., ZHOU, Y., NEWSOME, J., BRUMLEY, D., AND SONG, D. 2007. Sweeper: A lightweight end-to-end system for defending against fast worms. *SIGOPS Oper. Syst. Rev.* 41, 3, 115–128.
- US-CERT. Microsoft windows metafile handler setabortproc gdi escape vulnerability. <http://www.kb.cert.org/vuls/id/181038>.
- VAN OORSCHOT, P. C. 2003. Revisiting software protection. In *Proceedings of the Information Security Conference (ISC'03)*. 1–13.
- VIGNA, G., ROBERTSON, W., AND BALZAROTTI, D. 2004. Testing network-based intrusion detection signatures using mutant exploits. In *Proceedings of the ACM Conference on Computer and Communication Security (CCS'04)*. Washington, DC, 21–30.
- VULNERABILITIES 2006. <http://www.securityfocus.com/vulnerabilities>.
- WANG, H. J., GUO, C., SIMON, D. R., AND ZUGENMAIER, A. 2004. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'04)*. 193–204.
- WANG, K. AND STOLFO, S. J. 2004. Anomalous payload-based network intrusion detection. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID'04)*. 203–222.
- WANG, X., LI, Z., XU, J., REITER, M. K., KIL, C., AND CHOI, J. Y. 2006. Packet vaccine: Black-box exploit detection and signature generation. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'06)*. 37–46.
- WANG, X., PAN, C.-C., LIU, P., AND ZHU, S. 2006. Sigfree: A signature-free buffer overflow attack blocker. In *Proceedings of the 15th Conference on USENIX Security Symposium (SECURITY'06)*. Berkeley, CA, 215–240.
- WASSON, S. 2004. The NX bit. <http://techreport.com/reviews/2004q4/pentium4-570j/index.x?pg=1>.
- ACM Transactions on Information and System Security, Vol. 12, No. 2, Article 11, Pub. date: December 2008.

- WHYTE, D., KRANAKIS, E., AND VAN OORSCHOT, P. 2005. DNS-based detection of scanning worms in an enterprise network. In *Proceedings of the 12th Network and Distributed System Security Symposium (NDSS)*. 181–195.
- XU, J., NING, P., KIL, C., ZHAI, Y., AND BOOKHOLT, C. 2005. Automatic diagnosis and response to memory corruption vulnerabilities. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*. ACM Press, New York, NY, 223–234.
- YEGNESWARAN, V., GIFFIN, J. T., BARFORD, P., AND JHA, S. 2005. An architecture for generating semantics-aware signatures. In *Proceedings of USENIX Security Symposium (SECURITY'05)*.
- ZOU, C. C., TOWSLEY, D., GONG, W., AND CAI, S. 2005. Routing worm: A fast, selective attack worm based on ip address information. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation (PADS'05)*. 199–206.

Received February 2007; revised July 2007, September 2007; accepted September 2007