Feature Omission Vulnerabilities: Thwarting Signature Generation for Polymorphic Worms

Matthew Van Gundy, Hao Chen, Zhendong Su
University of California, Davis
{*vangundy,hchen,su*}*@cs.ucdavis.edu*

Giovanni Vigna
University of California, Santa Barbara
vigna@cs.ucsb.edu

Abstract

To combat the rapid infection rate of today's Internet worms, signatures for novel worms must be generated soon after an outbreak. This is especially critical in the case of polymorphic worms, whose binary representation changes frequently during the infection process.

In this paper, we examine the assumptions underlying two leading network-based signature generation systems for polymorphic worms: Polygraph [14] and Hamsa [12]. By identifying an assumption of both systems not met by all vulnerabilities, we discover a class of vulnerabilities (feature omission vulnerabilities) that neither system can accurately characterize. We demonstrate the limitations of Polygraph and Hamsa by testing the signatures that they generate for exploits targeting a feature omission vulnerability. We discuss why feature omission vulnerabilities are difficult to characterize and how increased semantic awareness can help the signature generation process.

1. Introduction

Internet-wide worm outbreaks in recent years have lead to the development of systems and techniques for limiting the spread of this type of malware. Content-based filtering has been shown to be effective for limiting the number of hosts infected by a worm outbreak [13], particularly when performed at points of high-connectivity, such as network routers. However, effective content filtering requires that signatures for new worms be generated and disseminated quickly. Since manual analysis of captured worm instances is too slow for this task, host-based and network-based systems for automated signature generation have been developed.

Unlike host-based systems, which use local knowledge about the vulnerable application or platform to generate signatures, network-based signature generation systems identify invariant portions of a worm's on-the-wire representation by comparing worm samples to samples of innocuous traffic. By only considering information available at the network level, network-based signature generation systems aim to apply to vulnerabilities in virtually any networked application.

Two leading network-based signature generation systems for polymorphic worms, Polygraph [14] and Hamsa [12], have demonstrated their ability to derive precise signatures for polymorphic exploits of a number of vulnerabilities. However, neither Polygraph nor Hamsa has discussed what type of vulnerabilities are amenable to automated signature generation using their techniques. By examining some of the assumptions underlying both systems, we identify a class of vulnerabilities that violate a common assumption. We then identify an example vulnerability from this class in the wild, and we show, through simulation, that both Polygraph and Hamsa are unable to derive precise signatures for simple exploits of this vulnerability.

The remainder of this paper is organized as follows. Section 2 covers related work including the high-level operation of Polygraph and Hamsa. Section 3 presents the assumptions common to both systems and describes a class of vulnerabilities which violates those assumptions. Section 4 describes our evaluation methodology. The results of our tests are described in Section 5. We then briefly discuss some of the implications of this class of vulnerabilities in Section 6. In Section 7 we discuss potential future work and present our conclusions.

2. Related Work

The problem of signature generation and filtering for monomorphic worms was first addressed by *content-sifting* systems such as Honeycomb [11], Autograph [10], and Earlybird [18]. After using heuristics to iden-

tify suspicious traffic, content-sifting systems attempt to find a single substring, usually on the order of 40 bytes in length, common to a significant fraction of the suspicious traffic. Using a single substring as a signature is appropriate for monomorphic worms since they do not change their representation between infections. However, this technique is inappropriate for polymorphic worms, which change significantly between infections, because there may be no byte-string of sufficient length common to a significant fraction of the worm instances.

In the case of polymorphic worms, network signature generation (or *pattern extraction*) systems rely on the assumption that there must be some invariant content present in each polymorphic worm instance to ensure that the worm can exploit the vulnerability successfully (for justifications see Section 3.1). Pattern extraction systems, such as Polygraph [14] and Hamsa [12], seek to generate signatures capturing these invariants. The architectures of Polygraph and Hamsa separate the problem of worm containment into three separate tasks: flow classification, signature generation, and content filtering.

The flow classifier is responsible for determining which network flows are likely to contain worms. It separates network traffic into two sets: innocuous and suspicious. Innocuous traffic is believed to be worm-free while suspicious traffic is believed to consist of worms in transit between hosts.

Once the suspicious flows have been identified, the signature generator attempts to derive a signature to match them. Both Polygraph and Hamsa begin by extracting all tokens (or substrings) longer than some minimum length that appear in a specified fraction of the suspicious network flows. Borrowing the formalism from Hamsa, the suspicious flow pool, denoted $\mathcal{M} = \{M_1, M_2, \ldots, M_{|\mathcal{M}|}\}$, is composed of suspicious flows (M_i) and the subset of flows in $\mathcal{M}$ which contain token t is denoted by $\mathcal{M}_{\{t\}}$. Given suspicious flow pool $\mathcal{M}$, a minimum token length l, and an extraction threshold λ, token extraction determines the set of tokens T such that $T = \{t \mid \forall_{M_i \in \mathcal{M}}\, \forall_{t \in M_i}.\ |t| \geq l \wedge \frac{|\mathcal{M}_{\{t\}}|}{|\mathcal{M}|} \geq \lambda\}$.

After token extraction, both systems attempt to derive signatures by choosing tokens that yield signatures that match a large fraction of suspicious flows while maintaining low false positive rates in the innocuous pool. The strategies used to derive signatures from the set of extracted tokens are specific to the system and the type of signature being generated.

Polygraph generates three types of signatures using these tokens: *conjunction*, *token subsequence*, and *Bayesian*. A flow matches a conjunction signature if it contains each token present in the signature. Likewise, a flow matches a token subsequence signature if it contains each token present in the signature in the same order that the token appears in the signature. Initially, Polygraph forms conjunction and token subsequence signatures for each individual suspicious flow by including all tokens occurring in the flow. Polygraph then generalizes these signatures by merging them greedily until a further merge would result in a signature with an unacceptable false positive rate in the innocuous pool. Finally, Polygraph outputs these resulting signatures.

Polygraph generates Bayesian signatures by calculating, for each token, the Bayesian score corresponding to the probability that the token will be found in a suspicious flow versus the probability that it will be found in an innocuous flow. A flow matches a Bayesian signature if the total score of all tokens found in the signature is greater than the matching threshold. Polygraph calculates the matching threshold as the lowest value that results in (at most) a given innocuous pool false positive rate.

Hamsa generates multi-set signatures similar to Polygraph's conjunction signatures except that each token has an associated frequency. For each token t_i and its associated frequency n_i, a flow must contain at least n_i occurrences of t_i in order to match the signature. Hamsa uses a model (Γ) which bounds the maximum false positive rate in the innocuous pool that a signature may have in terms of the number of tokens in the signature. Hamsa generates signatures greedily by eliminating all tokens that, when added to the current signature, would result in a false positive rate above the limit specified by Γ. Hamsa then chooses the remaining token (if any) that matches the largest number of flows in the suspicious pool. Hasma repeats this process until it can add no token to the signature that will satisfy Γ or it has reached the maximum number of tokens.

At the end of token selection, Hamsa scores the signatures from each step in the process according to their false positive rate and false negative rate in the innocuous and suspicious pools. Hamsa makes the signature with the highest score more specific by extending each token to the longest substring containing that token found in all suspicious pool flows matched by the signature. The signature generation process continues iteratively if a significant portion of the suspicious pool remains unmatched.

After signature generation completes, each system transfers its generated signatures to a network filter for intercepting worm flows in transit.

Pattern extraction systems like Polygraph and Hamsa identify patterns in network traffic without taking end host semantics into account. Several semantics-aware signature generation systems have been proposed, among them Vigilante [5] and the system described by Brumley et al. [2]. These systems leverage access to the applications running on the host to derive signatures capturing the semantics of the vulnerability being exploited. Because pattern extraction systems treat network streams as opaque byte strings, they are unable to take advantage of the richer semantic information available on the end host. However, this design decision also makes pattern extraction systems applicable to arbitrary end host configurations instead of being coupled to specific applications or platforms.

Concurrently with our work, Cui et al. [7] have proposed ShieldGen. ShieldGen uses network protocol analysis and a host-based exploit detection oracle (such as Vigilante) in order to derive an exploit signature expressed in terms of protocol constraints. Instead of attempting to generalize a signature from multiple exploit samples in the style of content-sifting systems, ShieldGen constructs new exploit instances by weakening different constraints and leveraging its oracle to determine which constraints are necessary for an exploit to succeed. This approach trades some of the semantic accuracy of a host-based solution for the greater generality of network-based solutions.

Perdisci et al. have demonstrated how to mislead Polygraph by injecting arbitrary amounts of specially-crafted noise into Polygraph's suspicious pool [16]. Newsome et al. extend Perdisci et al.'s attacks as well as showing how values, known as *red herrings*, can be employed to cause conjunction and token subsequence signatures generated by Polygraph to be overly specific [15]. Li et al. also propose an attack against Polygraph while describing the design motivation for Hamsa [12].

Our work is orthogonal to existing work in that we are not attacking these systems per se. Instead, we identify an assumption underlying the approaches of both systems that has not yet received scrutiny. In particular, we search for, and have identified, a class of vulnerabilities that do not fit the assumptions of either system.

3. Feature Omission Vulnerabilities

Signature generation techniques depend upon a number of assumptions about the setting in which they are employed and the threat model in question. Li et al. enumerate a number of these assumptions and discuss them with respect to Hamsa [12]. Briefly, they are:

Assumption 1: An attacker cannot control which worm samples are encountered by the system.

Assumption 2: An attacker cannot control which worm samples will be classified as suspicious by the flow classifier.

Assumption 3: An attacker cannot change the frequency with which tokens in normal traffic occur (innocuous pool poisoning).

Assumption 4: An attacker cannot control which innocuous flows will be classified as suspicious by the flow classifier (noise injection).

While Li et al. present Assumptions 3 and 4 as unique to Hamsa, they also apply to Polygraph. Perdisci et al. [16] and Newsome et al. [15] have shown how to violate these assumptions.

3.1. The Feature Addition Assumption

There are no assumptions in the above list regarding the type of vulnerabilities that are amenable to automatic signature generation. Both Polygraph and Hamsa assume that each worm sample must incorporate certain invariant tokens in order to successfully exploit the vulnerability that it targets. We formalize this in the following (additional) assumption which we will call the "Feature Addition Assumption".

Assumption 5: (Feature Addition) Each exploit of a given vulnerability must include one or more byte-strings not commonly found in innocuous traffic.

Hamsa formalizes its invariant uniqueness assumptions through its Γ model [12] while Polygraph only describes its uniqueness assumptions informally. Both systems, however, assume that each worm sample will include at least one element from some small set of byte-strings that do not commonly occur in innocuous traffic and, thus, are unique to malicious traffic.

This assumption is deeply embedded in the design of both systems. If we consider the method of token extraction used by Polygraph and Hamsa, we see that the set of tokens used to derive signatures consists entirely of substrings occurring in at least a λ-fraction of the flows

in the suspicious pool.

The justifications for the Feature Addition Assumption stem from traditional control-flow hijacking attacks. The Epsilon-Gamma-Pi model proposed by Crandall et al. describes the necessary components of a successful control-flow hijacking attack [6]. Informally, they are:

ϵ - represents protocol framing data that must be present in order to cause the control path of the application to reach the vulnerable point

γ - represents injected data which will be spuriously interpreted by the application as control data

π - represents the executable payload to which control will be transfered by γ

Because the payload (π) can be any executable code sequence, it can be highly variable. Crandall et al. argue that π is therefore not a good source of signature material for polymorphic worms. Consequently, both Polygraph and Hamsa focus on characterizing ϵ and γ – arguing that they should be sufficiently unique to distinguish worm samples from innocuous traffic.

The authors of both Polygraph and Hamsa argue that for most control-flow hijacking vulnerabilities there are a limited number of memory locations to which control can be transfered in order for an exploit to be successful. In this way, they argue that there are few potential values for γ and that γ is, therefore, a good potential source of signature material.

However, Crandall et al. note that γ can be highly variant due to prevalence of register springs[1] that may be targeted by an exploit. An exploit may also have considerable freedom with respect to the location where the exploit places its payload in memory. This can lead to further variability in γ. In light of these circumstances we believe that in general, like π, γ should not be considered a significant source of signature material.

In order to be successful an exploit must coerce the application to the vulnerable point. Thus, for ϵ, it is argued that since vulnerabilities usually lie on seldom used code paths, the protocol framing data needed to direct the application to the vulnerable point will be manifested as tokens unique to malicious traffic. For non-control-flow hijacking attacks [3] γ and π are empty, but as with control-flow hijacking attacks, the protocol framing data to direct the program to the vulnerable point (ϵ) must be present. Hence, it is argued that methods such as Polygraph's and Hamsa's are able to isolate these unique ϵ tokens, making these systems applicable to both control-flow hijacking and non-control-flow hijacking attacks alike.

[1] A *register spring* is an opcode sequence which will transfer control to the memory address denoted by a register. For instance, returning control to a memory location which corresponds to the opcode `jmp %esp` will cause execution to be returned to the stack without requiring the address of the payload on the stack to be included in γ.

3.2. Violating the Feature Addition Assumption

The Feature Addition Assumption is a constraint on vulnerabilities rather than on an attacker's abilities or the deployment setting. Any vulnerability for which the Feature Addition Assumption does not hold will fall outside the scope of both systems. Yet, the Feature Addition Assumption appears to be trivially violable.

For example, suppose an application can be moved to a vulnerable point in its control path by omitting a piece of data required by its network protocol. In this case ϵ denotes the absence of some token in an expected location. This results in a malicious protocol framing feature set that is a subset of the innocuous protocol framing feature set. This is a concept that the algorithms of neither Polygraph nor Hamsa capture. Both systems obtain the set of tokens from which they derive signatures solely from malicious traffic, which does not contain the absent token.

Even if token extraction were somehow able to capture, as a feature, that such a token was omitted, this scenario cannot be expressed by the non-Bayesian signature models of either system. In both systems, all non-Bayesian signatures match a flow only if the flow contains all of the tokens present in the signature. There is no provision for a signature to match only flows that *do not* contain a token. We denote vulnerabilities of this type by the term *feature omission vulnerabilities* indicating that some *feature* common to innocuous traffic has been omitted from malicious traffic in order to exploit the vulnerability. We hypothesize that Polygraph and Hamsa will not be able to derive precise signatures for a feature omission vulnerability that allows sufficient variability in γ.

3.3. An Example from the Wild

While a hypothetical class of vulnerabilities falling outside the scope of current pattern extraction systems is an important discovery, we wished to verify through practice that feature omission vulnerabilities could indeed thwart signature generation in both Polygraph and Hamsa. Most vulnerabilities are discovered on seldom-used control paths. These paths could be seldom taken

because they may be associated with a seldom-used protocol feature, or because most client applications do not cause the invocation of code handling exceptional protocol conditions.

While surveying vulnerabilities in CVE [4] and Bugtraq [17] we discovered CVE-2004-0597 [8], which describes a stack-based buffer overflow vulnerability in the way libpng, versions 1.2.5 and earlier, processes PNG images. This vulnerability had wide impact. A brief survey of its Bugtraq entry (BID 10857) shows that it affected system and application software alike on more than four operating system families.

Portable Network Graphics (PNG) files are formatted as a series of discrete chunks of various types. The type of a chunk denotes its purpose. The PNG Specification [1] imposes a partial ordering on the chunks that compose a PNG image file. A chunk of type PLTE containing color palette entries must be present in every indexed-color image. The PLTE chunk may (optionally) be followed by a chunk of type tRNS, which specifies transparency information for each of the palette entries. However, if a tRNS chunk is present it *must* appear after the associated PLTE chunk.

Before version 1.2.5, libpng contained a buffer overflow vulnerability [9] that could be triggered by supplying an indexed-color image that omitted the required PLTE chunk. This causes the routine that decodes the tRNS chunk to bypass a length check and write an arbitrary amount of data into a fixed-length stack buffer. In the terms of the $\epsilon\gamma\pi$ model:

- ϵ - The PNG image must be of indexed-color type and the PLTE chunk required by the specification must not occur before the tRNS chunk carrying the payload.
- γ - There are a number of possible exploitations of this vulnerability. However, the most straightforward exploits would use a stack-based return address or a register-spring address to overwrite the tRNS chunk handler return pointer.
- π - The only constraint on the payload is that it must be longer than 256 and shorter than 2^{31} bytes.

With respect to ϵ, this vulnerability falls within the class of feature omission vulnerabilities. The necessary precondition for exploitation is the absence of the PLTE chunk. Furthermore, because of the structure of a PNG file, omitting the PLTE chunk is unlikely to cause adjacent chunks to be combined to form a new feature. Also, while the control data (γ) that must be injected into the application must be taken from a limited set of valid values, we hypothesized that the size of γ was likely to be large enough to resist generation of precise signatures. The likelihood that values in γ will be coincidentally found in innocuous flows is also heightened because PNG's encapsulate compressed binary data.

4. Evaluation

To test our hypothesis that Polygraph and Hamsa are unable to accurately characterize worms targeting feature omission vulnerabilities, we generated a number of simulated worm samples targeting the libpng vulnerability. We then replicated the experimental setups of Polygraph and Hamsa as faithfully as possible and tested the accuracy of the signatures generated by each system on our worm samples.

4.1. Exploit Generation

We created exploit generators for the two most straight-forward methods of exploiting the libpng vulnerability: via stack-based return address and via register spring. Each exploit consists of an executable payload and control data to direct execution to that payload wrapped in a minimal PNG file that has all non-essential fields chosen uniformly at random.

For the return-to-stack exploit, we exhaustively generated variants that explore every possible location of the payload on the stack, yielding 3,937 working variants. We positioned the 4-byte return address of the payload so that it would overwrite the return pointer and used bytes chosen uniformly at random for padding and alignment.

For the register spring exploit, we searched the vulnerable application and shared libraries for byte sequences that form a `jmp %esp` (or equivalent) instruction. Even in our minimal test application, which merely opens a PNG file and feeds it to libpng, we discovered 35 such springs at various locations in the application and shared libraries. It bears mentioning that, due to the small size of the application, all register spring addresses begin with the byte \xb7 and 77% of the addresses have either \xf7 or \xf8 as their second byte. In order to offset the bias towards these two 2-byte prefixes, we weighted the probability that each address would be used in order to achieve a distribution of 2-byte substrings that was as close to uniform as possible.

For the register spring exploit to be successful, we also needed to increment (lift) the stack register by 8 bytes. We found many suitable opcodes in the program text (far more than the number of suitable register

Table 1. Parameters for Polygraph and Hamsa Evaluations

Polygraph Parameters	
minimum token length	2 chars
token extraction threshold	3 flows (20% for Bayes)
minimum cluster size	3 flows

Hamsa Parameters	
minimum token length (l)	2 chars
token extraction threshold (λ)	0.15
maximum false positive rate ($u(1)$)	0.15
false positive reduction factor (u_r)	0.5
maximum signature tokens (k^*)	15

springs). When constructing the register spring exploits, we chose lift addresses using the same technique used to avoid biasing in the register spring addresses.

After generating the exploits, we verified that each one successfully exploited the vulnerable application. We then overwrote the payload portion of the exploit with random bytes to simulate a perfectly polymorphic payload in a manner similar to Polygraph's experiments [14]. In this way, we left intact all protocol framing (ϵ) and control data (γ) while simulating the significant amount of freedom available in constructing a polymorphic payload (π).

4.2. Data Collection and Innocuous Pool Creation

To create an innocuous traffic pool, we crawled the web directories of a campus web server and meta-searched search engine results for PNG images. In all, we amassed over 147,000 images. About 38% of these are indexed-color images, the same type used for the exploits. We reassembled the network traces of these images being fetched from a web server and partitioned them into a 504MB training pool and a 5.4GB test pool.

As systems that attempt to characterize misuse, Polygraph and Hamsa's accuracy should not be highly sensitive to the composition of the innocuous training pool, otherwise their accuracy would become unpredictable in practice. Even so, we did not include other HTTP traffic in the innocuous pool in order to provide Polygraph and Hamsa with an optimal scenario, in which they learn signatures based on a maximally good innocuous pool and a maximally bad suspicious pool.

4.3. Experimental Setup

To help ensure that our results are as comparable as possible to the original publications, we used parameter values (Figure 1) equivalent to those specified in the Polygraph and Hamsa papers [14, 12]. For both exploits, we ran 5 signature generation trials for suspicious training pools of each of the following sizes: 5, 10, 25, 50, 100, 200. For each exploit, we collected the remaining malicious flows into a test pool, used to determine the false negative rate of the generated signatures.

To help ensure that the systems were appropriately tuned, we ran the test sets for each exploit against Hamsa with 300 different combinations of parameter values. The best results achieved were identical to those obtained using the parameters in Figure 1. We did not conduct similar trials with Polygraph due to its significantly longer running time. However, our experience with Hamsa gives us some confidence that the results reported in the next section show these systems in a fair light.

5. Results

For the pattern-based signatures (i.e. Conjunction, Token Subsequence, and Hamsa) the median results for all trials were no false positives and 100% false negatives, that is, no worm instances were correctly detected. The false positive and false negative standard deviation was 0% for the return-to-stack exploits. For the register spring exploits, the false positive standard deviation was less than 1.10%, and the false negative standard deviation was less than 6.92% in each trial.

Figure 1. Best Token Subsequence Signature

```
('HTTP/1.0 200 OK\r\nConnection: keep-alive\r\nCont
ent-Type: image/png\r\nETag: "', '"\r\nAccept-Range
s: bytes\r\nLast-Modified: ', ' GMT\r\nContent-Leng
th: ', '\r\nDate: Mon, 19 Mar 2007 21:1', ' GMT\r\n
Server: lighttpd/1.4.11\r\n\r\n\x89PNG\r\n\x1a\n\x00
\x00\x00\rIHDR', '\x03\x00\x00', '\x00\x00', 'tRNS',
'IEND')
```

The best Token Subsequence and Hamsa signatures (Figures 1 and 2) were generated for the register spring exploit. The best Conjunction signature is identical to

Figure 2. Best Hamsa Signature


```
{'\x00\x00': 4, 'en': 3, 've': 2, 'ag': 2, ' M': 2,
'Mo': 2, ' GMT\r\nServer: lighttpd/1.4.11\r\n\r\n
\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR': 1, ': ': 8,
'ge': 2, '\x03\x00\x00': 1, 'li': 2, 'e: ': 2,
'ep': 2, 'es': 2, 'er': 2, ', ': 2, ' 2': 4,
'\r\nContent-': 2, '/1.': 2, 'on': 5, ' 200': 3,
'"\r\nAccept-Ranges: bytes\r\nLast-Modified: ': 1,
' GMT\r\nContent-Length: ': 1, 'ng': 3, '0 ': 2,
'HTTP/1.0 200 OK\r\nConnection: keep-alive\r\n
Content-Type: image/png\r\nETag: "': 1, 'te': 4,
'IEND': 1, '\r\nCon': 3, ' GMT\r\n': 2, 'nt': 4,
'\r\nDate: Mon, 19 Mar 2007 21:1': 1, '\r\n': 11,
't-': 4, 'tRNS': 1}
```


the Token Subsequence signature in Figure 1 with the omission of the `\x00\x00` token. The false negative rate for all three signatures was 82.7%.

If we examine these signatures, we see that there are no tokens corresponding to π (the polymorphic payload). Because we simulated a perfectly polymorphic payload, any common payload tokens between instances are purely coincidental. Several tokens from ϵ (the protocol framing data) appear, including tokens indicating indexed-color (`\x03\x00\x00`) PNG images (`...\x89PNG...` and `Content-Type: image/png`) and the presence of a tRNS chunk. The token `PLTE` is absent from the signatures because a PLTE chunk is not included in the exploits. However, according to the signature models for these signatures, the absence of a `PLTE` token indicates that the token may or may not be present. None of the signatures are able to express the condition that the PLTE chunk must be *absent*.

The γ (bogus control) data from the register spring vulnerabilities (e.g. \xf7\xb7, \xf8\xb7, etc.) does not appear in any of the signatures, confirming our hypothesis that the degree of variation available to register spring exploits, even in our small vulnerable application, is sufficient to prevent precise recognition of γ by Polygraph and Hamsa's algorithms. The top two bytes of the stack-based return address (\xff\xbf) are present in all of the signatures for the return-to-stack exploit. However, because PNG images consist primarily of compressed data having uniform character distribution, \xff\xbf is found in about 63% of the innocuous training flows. A two-byte invariant, such as this, may be sufficient to detect binary exploits embedded in a text-based protocol, but it is not a good discriminator against a background of binary image data.

In addition to tokens taken from ϵ and γ, there are a number of other tokens present that are portions of HTTP headers included in all responses from the web server software used to deliver the exploits. However, Polygraph and Hamsa are unable to determine that these tokens are not important because they do not consider protocol semantics. The tokens responsible for many of the false negatives for these signatures fall into this category.

Because the libpng vulnerability we tested is a feature omission vulnerability, the small amount of data gleaned from ϵ and γ is more common in the innocuous training pool than the timestamps in the Date headers. Because the innocuous training pool must be collected before a worm outbreak begins, the timestamps present in training flows will be disjoint with any legitimate timestamps found in suspicious flows. This causes the timestamp data from suspicious flows to be incorporated into the signatures, limiting a signature's maximum usefulness to a window of 10 minutes for the register spring exploits and 1 minute in the case of the return-to-stack exploits. This occurs because Polygraph and Hamsa do not consider protocol semantics and therefore cannot determine that the Date header is not necessary for the exploit to succeed. This problem cannot be remedied by simply omitting these Date tokens without raising the false positive rate due to the systems' inability to precisely characterize ϵ and γ.

One of Hamsa's features exacerbates this issue, further degrading the accuracy of its signatures. Some of the initial signatures generated by Hamsa were much less specific and would have matched more exploits. However, after generating a signature, Hamsa refines the signature by finding all tokens in the covered portion of the suspicious pool which match the initially generated signature. It includes all of these (possibly longer) tokens into the signature, making the signature more specific, thereby reducing the possibility of false positives while maintaining the same coverage in the suspicious pool. In a few of the cases, this caused a signature with very little timestamp information to be expanded into one that included a maximal amount of timestamp information and, therefore, had a minimal chance of catching future exploits.

The median false positive rate for Polygraph's Bayesian signatures for the return-to-stack exploit on all pool sizes was less than 0.12% with a standard deviation of less than 0.06%. For the register spring exploit, the Bayesian signatures for all pool sizes had less than 0.05% false positives with a standard deviation of less than 0.69%. As Figure 3 shows, the median false negative rates for Polygraph's Bayesian signatures were lower than those of the pattern-based signatures. The maximum standard deviation for any of the pool sizes

Figure 3. Bayesian Signature Performance

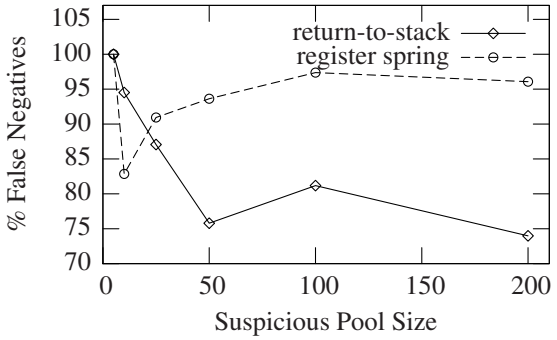

Figure 4. Best Bayesian Signature

```
{'\x08\x03\x00\x00': 0.0017, ':5': 0.0042,
'7 ': 0.0073, '0 ': 0.0097, '11': 0.0120,
'at': 0.0141, ' 0': 0.0159, '8 GMT\r\n': 0.0208,
'6 GMT\r\n': 0.0422, 'n, ': 0.0714, 'HTTP/1.0 200 OK
\r\nConnection: keep-alive\r\nContent-Type: image/png
\r\nETag: "-': 0.0337, '5 GMT\r\n': 0.0435,
'9 GMT\r\n': 0.0723, ':2': 0.0837, 'IEND': 0.1107,
' GMT\r\nServer: lighttpd/1.4.11\r\n\r\n\x89PNG\r\n
\x1a\n\x00\x00\x00\rIHDR': 0.1113, '\xff\xbf': 0.1133,
'3 ': 0.1152, ' GMT\r\nContent-Length: 1': 0.1341,
'HTTP/1.0 200 OK\r\nConnection: keep-alive\r\nContent
-Type: image/png\r\nETag: "-1': 0.1798, 'HTTP/1.0 200
OK\r\nConnection: keep-alive\r\nContent-Type:
image/png\r\nETag: "1': 0.1856, ' 2006 ': 0.3172,
'"\r\nAccept-Ranges: bytes\r\nLast-Modified:
S': 0.2192, ' GMT\r\nContent-Length: 2': 0.2573,
'"\r\nAccept-Ranges: bytes\r\nLast-Modified: Wed,
': 0.2767, ' GMT\r\nContent-Length: 3': 0.4708,
'\x04\x03\x00\x00': 0.4841, ' 2007 1': 0.7817,
'\x02\x03\x00\x00': 0.7230, 'tRNS': 0.5656,
'\x01\x03\x00\x00': 0.7352, ', 2': 0.7460,
'\x03\x00\x00\x01': 0.9785, ' 2005 ': 0.3139,
'\r\nDate: Fri, 23 Mar 2007 07:27:4': 7.7908,
'\r\nDate: Fri, 23 Mar 2007 07:27:5': 8.2823,
} Threshold: 3.4869
```

was less than 10%. However, even the best Bayesian signature (depicted in Figure 4) had a false negative rate of 63.11%, severely limiting its usefulness for effectively mitigating a worm outbreak. This signature, generated for the return-to-stack exploit, incorporates the supersets of many of the tokens found in the pattern-based signatures, and, like the pattern-based signatures, it includes the Date headers from the exploit flows. In fact, the Bayesian scores for the Date tokens are above the matching threshold and are nearly an order of magnitude greater than `\x03\x00\x00\x01`, the first three bytes of which are part of ϵ.

Because we simulated a perfectly polymorphic payload (π), these signatures must capture the essential protocol framing (ϵ) and control (γ) data in order to correctly classify the exploits. The signatures must either recognize the presence of a tRNS PNG chunk without a preceding PLTE chunk (ϵ) or the presence of stack or register spring return addresses (γ), or both. However, our results demonstrate that both Polygraph and Hamsa are unable to correctly characterize the protocol framing data (ϵ) of the selected feature omission vulnerability and, in the face of highly-variant control (γ) and payload (π) data, they are mislead by accessory protocol data because they cannot use protocol semantics to determine which tokens are unimportant.

6. Discussion

One reason why Polygraph and Hamsa's cannot characterize feature omission vulnerabilities is that they cannot recognize omitted features because they do not attempt to characterize innocuous traffic. However, the solution is not straightforward. Characterizing innocuous traffic precisely appears to be a much harder problem than characterizing malicious traffic. Finding a set of features common to worm instances targeting the same vulnerability is probably much easier than finding a set of common features among a diverse set of innocuous traffic that may vary significantly in type, purpose, and importance. For many protocols, a set of features common to all innocuous traffic is likely to be so minimal as to be useless.

Both systems are also easily mislead by unimportant protocol features because their context-agnostic design prevents protocol or application specific information from being used to determine a feature's importance. While these systems aim to automate a job that a human analyst performs by manually inspecting worm instances for invariant signature material, the human analyst is likely to have the contextual advantage of understanding the network protocols in play. The human analyst can use this knowledge to rank the importance of different features shared by a family of exploits. Without the benefit of context, these systems can probably do little better than their current approach of ranking features based solely on their rates of occurrence.

Even if Polygraph and Hamsa could recognize the omission of a feature, they may not be able to determine if the feature has been omitted without full knowledge of the protocol in question. For instance, assume that a pattern extraction system could determine that a PNG image containing a tRNS chunk without a preceding PLTE chunk was a sufficient condition for exploit-

ing the libpng vulnerability. That system could not determine that the PLTE chunk was missing without full knowledge of the PNG format because the worm could always embed a fake PLTE chunk within the bounds of another chunk.

The development of ShieldGen [7], concurrently with this work, provides some insight into the value of increased semantic awareness. ShieldGen uses a network protocol analyzer to correctly parse network streams and develop signatures based on the structure and values of the individual protocol fields. ShieldGen also leverages knowledge of the purpose of various fields to prevent it from being misled easily by inherent red herrings (such as timestamps). We believe that ShieldGen may be capable of generating effective signatures against the libpng vulnerability using its additional knowledge of protocol syntax and semantics.

The libpng vulnerability is a simple buffer overflow. ShieldGen employs a buffer overflow heuristic which should detect that exploits only succeed when a tRNS chunk with a length of more than 256 bytes appears. This signature would catch all exploits with no false positives among correctly formatted PNG files. However, if the vulnerability was not a simple buffer overflow, or depended on the contents of multiple fields, this heuristic would not succeed in finding an effective signature.

ShieldGen also determines if the exploit flow violates any protocol constraints. If so, it attempts to construct probes that meet the constraints to determine if the exploit is triggered by violating one or more of those constraints. In this phase, ShieldGen may be able to recognize that the absence of the PLTE chunk preceding the tRNS chunk violates the PNG format. If ShieldGen were to issue a probe by adding a PLTE chunk, it would learn that the presence of a tRNS chunk without a preceding PLTE chunk are necessary conditions for the exploit to occur. With this knowledge, ShieldGen could generate a precise vulnerability signature. However, if instead, ShieldGen generated a valid probe by dropping the tRNS chunk and noting that the resulting flow does not successfully exploit the vulnerability, it could falsely conclude that the presence of the tRNS chunk is the sufficient condition for the exploit to occur.

While an improvement over current pattern extraction systems, ShieldGen's ability to recognize this feature omission vulnerability in terms of the omitted feature depends on the way in which it generates probes. Because ShieldGen does not exhaust all combinations of protocol features, it is not clear that ShieldGen can catch feature omission vulnerabilities in general. Furthermore, if the omitted feature did not cause a violation of the protocol, it is unlikely that ShieldGen would be able to generate a precise signature. This is because ShieldGen's initial signature consists of all protocol constraints output by the protocol analyzer and it derives a signature by relaxing or omitting constraints. If a constraint representing the absence of the feature is not output by the protocol generator initially, it will not be considered during ShieldGen's probing process.

7. Conclusions and Future Work

The wide variety of Internet devices makes host-agnosticism a desirable feature for signature generation systems. While there are many different types of hosts, and many more possible configurations, every pair of communicating hosts must share some common set of protocols. ShieldGen serves as a positive example of how incorporating knowledge of protocol syntax and semantics can aid the generation of precise signatures while still remaining relatively host-agnostic.

Having more feature data alone will not solve the problem of generating precise signatures for feature omission vulnerabilities. Better techniques for recognizing the necessary conditions for an exploit to succeed must be developed. For example, using increased knowledge of the protocols in play, a system might be able to cluster similar types of innocuous and suspicious traffic together, and then generate more precise signatures by noting the features present in innocuous traffic but absent from the suspicious traffic within the cluster. Polygraph and Hamsa already do a very primitive sort of clustering along these lines by partitioning traffic by service port. However, any signature that mistakenly assumes that some feature(s) must be omitted risks trivial evasion by a worm that includes some subset of those features as red herrings. In some situations it may be impossible to distinguish the two situations without full knowledge of how the application will process the data.

Lastly, regardless of the techniques used for signature generation, the signature models employed must be able to express the notion that some feature must be absent in order for the signature to match.

We have evaluated the operation of two of the leading pattern extraction systems for polymorphic worms, and we have uncovered a class of vulnerabilities that violate an assumption of these systems – *feature omission vulnerabilities*. We discovered an instance of a feature omission vulnerability in the wild and implemented exploit generators for this vulnerability in order to test Polygraph's and Hamsa's ability to generate meaningful signatures for it. We demonstrated that Polygraph and

Hamsa are ineffective against attacks that exploit this type of vulnerability.

Acknowledgments

This research was partially supported by the National Science Foundation, under grants CCR-0238492, CCR-0524853, CCR-0716095, and CNS-0644450, by US Air Force under grant FA9550-07-1-0532, and by the University of California, Davis TOPS Fellowship.

References

[1] M. Adler, T. Boutell, J. Bowler, C. Brunschen, A. M. Costello, L. D. Crocker, et al. Portable Network Graphics (PNG) Specification (Second Edition). Technical report, W3C, Nov. 2003. http://www.w3.org/TR/2003/REC-PNG-20031110.

[2] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards Automatic Generation of Vulnerability-Based Signatures. In *IEEE Symposium on Security and Privacy*, Washington, DC, USA, May 2006. IEEE Computer Society.

[3] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In *USENIX Security Symposium*, pages 177–192. USENIX The Advanced Computing Systems Association, Aug. 2005.

[4] Common Vulnerabilities and Exposures. http://cve.mitre.org/.

[5] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of internet worms. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 133–147, New York, NY, USA, Oct. 2005. ACM Press.

[6] J. R. Crandall, Z. Su, S. F. Wu, and F. T. Chong. On Deriving Unknown Vulnerabilities from Zero-Day Polymorphic and Metamorphic Worm Exploits. In *ACM Conference on Computer and Communications Security (CCS)*, New York, NY, USA, Nov. 2005. ACM Press.

[7] W. Cui, M. Peinado, H. J. Wang, and M. E. Locasto. ShieldGen: Automatic Data Patch Generation for Unknown Vulnerabilities with Informed Probing. In *IEEE Symposium on Security and Privacy*, Washington, DC, USA, May 2007. IEEE Computer Society.

[8] CVE-2004-0597: Multiple buffer overflows in libpng 1.2.5 and earlier. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-0597.

[9] C. Evans. CESA-2004-001: libpng 1.2.5 stack-based buffer overflow and other code concerns, 2004. http://scary.beasts.org/security/CESA-2004-001.txt.

[10] H.-A. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *USENIX Security Symposium*, pages 271–286. USENIX The Advanced Computing Systems Association, Aug. 2004.

[11] C. Kreibich and J. Crowcroft. Honeycomb: Creating Intrusion Detection Signatures Using Honeypots. *SIGCOMM Comput. Commun. Rev.*, 34(1):51–56, 2004.

[12] Z. Li, M. Sanghi, Y. Chen, M.-Y. Kao, and B. Chavez. Hamsa: Fast Signature Generation for Zero-day Polymorphic Worms with Provable Attack Resilience. In *IEEE Symposium on Security and Privacy*, pages 32–47, Washington, DC, USA, May 2006. IEEE Computer Society.

[13] D. Moore, C. Shannon, G. M. Voelker, and S. Savage. Internet Quarantine: Requirements for Containing Self-Propagating Code. In *INFOCOM*, Apr. 2003.

[14] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *IEEE Symposium on Security and Privacy*, pages 226–241, Washington, DC, USA, May 2005. IEEE Computer Society.

[15] J. Newsome, B. Karp, and D. Song. Paragraph: Thwarting Signature Learning by Training Maliciously. In D. Zamboni and C. Kruegel, editors, *Recent Advances in Intrusion Detection (RAID)*, volume 4219 of *LNCS*, pages 81–105, New York, NY, USA, Sept. 2006. Springer-Verlag.

[16] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. Sharif. Misleading Worm Signature Generators Using Deliberate Noise Injection. In *IEEE Symposium on Security and Privacy*, pages 17–31, Washington, DC, USA, May 2006. IEEE Computer Society.

[17] Security Focus Vulnerability Notes (Bugtraq Database). http://www.securityfocus.com/vulnerabilities.

[18] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated Worm Fingerprinting. In *Operating Systems Design & Implementation (OSDI)*, pages 45–60. USENIX The Advanced Computing Systems Association, Dec. 2004.