Tales from the Crypt: fingerprinting attacks on encrypted channels by way of retainting

Michael Valkering, Asia Slowinska, Herbert Bos

Department of Computer Science

Faculteit der Exacte Wetenschappen

Vrije Universiteit Amsterdam

De Boelelaan 1081

1081 HV Amsterdam, Netherlands

{mjvalker, asia, herbertb}@few.vu.nl

Abstract. Paradoxically, encryption makes it hard to detect, fingerprint and stop exploits. We describe *Hassle*, a honeypot capable of detecting and fingerprinting monomorphic and polymorphic attacks on encrypted channels. It uses dynamic taint analysis in an emulator to detect attacks, and it tags each tainted byte in memory with a pointer to its origin in the corresponding network trace. Upon detecting an attack, we correlate tainted memory blocks with the network trace to generate various types of signature. As correlation with encrypted data is difficult, we retaint data on encrypted connections, making tags point to decrypted data instead.

1 Introduction

Intended to enhance the security of network communication, encryption also makes it harder to detect and analyse attacks on the Internet. Strong encryption and pacing on network links lead to traffic that is more or less uniformly distributed in space and time, preventing the extraction of useful information. Methods relying on the observation of traffic characteristics no longer work. Examples include Snort and Bro that use byte patterns [15, 21], analysis of executable code in the network [17], static analysis techniques [7, 24] when applied in the network, and analysis of protocol fields [10, 11]. In addition, while advanced honeypot systems like Vigilante [12], TaintCheck [14] and Argos [18] would *detect* attacks, most common techniques for signature generation cannot be directly applied. Paradoxically, the very nature of encryption may turn against the original security goals.

At the same time, the use of encryption is increasing in almost all network services, including file systems, web servers, VPNs, databases, p2p, instant messaging, etc. Rather than considering the fairly narrow set of exploits against encryption libraries themselves (like Linux' Slapper [16], and Windows' SSL Bombs [1]), this work is motivated by the larger class of attacks that exploit the applications *using* encryption. It is well-known that given a choice between port 80 (http) and port 443 (https), attackers tend to opt for 443 almost without exception [19]. The reason is that the content of these channels cannot be so easily inspected by firewalls and virus scanners.

Instead of providing a NIDS with copies of the servers' private keys [13], something administrators may be reluctant to do, we prefer to push fingerprinting to the end-application on the host. On the other hand, we do not want to code manually a specific solution for each application. Rather, we are interested in methods for signature generation that apply to a wide variety of applications.

We emphasise that channel encryption should not be confused with polymorphism. Even though encryption yields unique network appearances for all network attacks, the nature of the attacks (polymorphic or not) still surfaces after decryption.

This paper discusses *Hassle*, a honeypot capable of detecting and fingerprinting monomorphic and polymorphic attacks on SSL-encrypted channels. *Hassle* is not application-specific and can be applied to any process that uses SSL for secure communication. We discuss both its design and its implementation on an x86 Linux-based architecture.

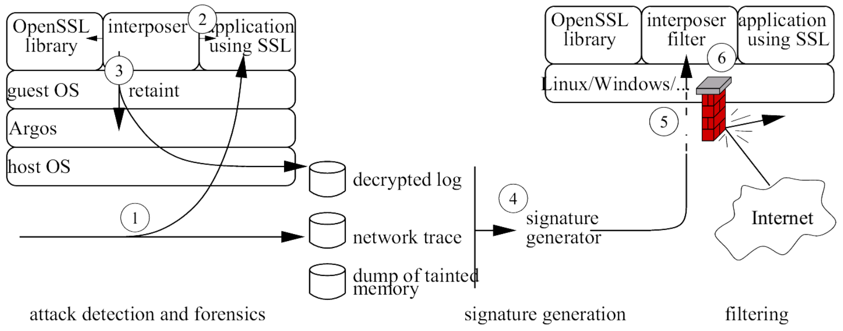

Fig. 1: Interposition, retainting, and signature generation.

SSL encryption. Encryption can be applied at many layers in the protocol stack. The most common examples in practice include the data-link layer (WEP, WPA), the network layer (IPSec), and the application (SSL). As layer-2 encryption in the NIC reduces the problem to that of non-encrypted channels at the OS level and can therefore be handled easily by emulators with dynamic taint [18], the most interesting design alternatives to consider in practice concern IPSec and SSL. Without loss of generality, we opted for implementing *Hassle* for SSL, as it is supported by many servers. Nevertheless, the same techniques can be applied at other layers.

Contribution. To the best of our knowledge, we are the first to address the problem of signature generation (and attack filtering) for encrypted communication, while also handling non-encrypted channels. The techniques we describe are applicable to most types of encryption and require no modification of the applications that need protection. In addition, we describe various novel signature generators that combine with *Hassle*. Besides well-known Snort-like signatures [21], we generate very accurate signatures for *polymorphic* buffer overflows on heap or stack.

The remainder of this paper is organised as follows. Sections 2 and 3 discuss architecture and implementation, respectively. *Hassle* is evaluated in Section 5. In Section 6, we discuss related work, while conclusions are drawn in Section 7.

2 Architecture

At the highest level, our system consists of a detection engine, a signature generator, and a filter, as illustrated in Figure 1. The detector is a honeypot based on a full-system hardware emulator that provides taint analysis. For this purpose, we modified an existing honeypot, known as Argos [18].

Assume for now that no encryption is used. All data from the network is logged to a rolling trace file (1). By means of taint analysis, *Hassle* tags and then tracks network data throughout the system (1), where a tag points to the origin of the data in the network trace. Whenever tainted data is used in a way that violates the security policy, we raise an alarm. Examples of such behavior include attempts to execute tainted data. At that point, *Hassle* dumps as much relevant data to disk as possible. For instance, for the process or kernel under attack, we save all tainted memory blocks with their tags, the names of the executable and the libraries used, and the address that triggered the alert together with its origin.

The signature generator correlates the data dumped by *Hassle* with those of the network trace to determine a signature (4). For instance, one of our signature generators dissects the input stream to determine which protocol fields were responsible for a buffer overflow and computes an upperbound on the combined length of the protocol fields as a signature. Any message in which the length of these protocol fields exceeds the upperbound is guaranteed to result in an overflow. We use the signature to block the attack elsewhere in the network without needing heavy-weight instrumentation (5).

Unfortunately, in case of encryption correlation between network trace and memory dump is not possible as all memory tags point to seemingly meaningless, uniformly distributed bytes in the network trace (2).To solve this we want to restore a meaningful correlation, albeit not to the network trace directly. For encrypted channels we *retaint* the tagged data after decryption (3). Concretely, we use library interposition to place a small amount of code between the application and the encryption library. The interposer requests the emulator to retaint data using offsets in the decrypted streams as tags. The decrypted data is stored in a log for future use. Although interposing leads to exposure of private data, we regard this (confined) exposure as less threatening than the possibility that the system is completely compromised by an attack via that data.

Signature generation (4) now progresses much like that of non-encrypted channels, albeit at a higher level in the protocol stack. When working at this level, well above the transport layer, we cannot simply dissect the network data stream from the first network packet onwards to determine the protocol fields that were used in the attack. Instead, we use the retainted decrypted network stream. Similarly, the filters that block traffic that contains the signature also must operate at higher-level protocol units (6). We implement them as *interposer filters* between the SSL library and the application that flag or drop all traffic towards the application that matches the signature.

Before we start discussing the precise nature of our signatures, we mention that as much as possible we focus on exploits rather than payloads, for several reasons. First, exploits exhibit fewer opportunities for polymorphism. Second, one exploit is often used for different payloads and stopping it kills multiple birds with one stone. Third, blocking exploits prevents many attacks from entering the system altogether.

2.1 Tracking issues

Tagging and tracking conceptually consists of adding meta-data to every byte in memory. The more meta-data is added, the more powerful the attack analysis can be. Again, we first consider the case that no encryption is used, and subsequently address encrypted channels in Section 2.2.

In *Hassle*, we allow three types of tagging. The cheapest, (incurring an overhead of approximately 15× on average), but also the weakest, is known as *black-white tagging* and simply indicates for each byte whether it originated in the network. It provides no clues as to the origins of tainted data in memory other than that it came from a suspect source. The tag is a single bit and no immediate correlation of network trace and memory is possible. It may be possible to align patterns in memory and network trace by means of similarity search [10, 18], but the margin of error is large.

A more powerful tagging method is known as *net tracking*, which keeps track of the network origin of tainted memory. In *Hassle*, we have implemented two modes of net tracking: *full origin*, and *single origin*.

Full-origin tracking is the most precise form of net tracking, but also the slowest (with a slowdown of about two orders of magnitude). Whenever data arrives from the network we tag it with a pointer to the corresponding bytes in the network trace. Whenever two tainted values are combined (e.g., an addition of two tainted locations), we retain the tags of both of them. This is implemented by maintaining a set of origin pointers that refer to the tags of the (one, two or three) tainted operands that produced this data. By applying the procedure recursively whenever tainted values are combined, we construct a tree with leaves pointing into the actual network trace. In practice, the amount of memory needed for the administration is approximately three origin pointers per tainted word. The overhead of maintaining the origin pointers is also considerable.

In contrast, single-origin tracking retains a single origin pointer that points to a byte in the network trace directly. If two tainted values are combined, we pick one of the tags for the destination. Single-origin tracking introduces some imprecision in the tracking. In practice, however, we have not seen instances where such imprecisely tagged data can be exploited by attackers. The advantage of single-origin is that it is much more efficient both in memory (one word per word of tainted data) and in computation (reducing the overhead to less than 20×). For this reason, we have used single-origin tracking for this paper. If the nature of applications changes such that single-origin tracking becomes an issue, we can switch to full-origin tracking in the future.

The strongest tagging method, known as *age-stamped net tracking*, maintains not only (full or single origin) net tracking, but also age stamps per tainted value. The age stamps serve to separate different buffers on the stack or the heap. For instance, every function call results in a new age stamp, and all tainted stores in the function are associated with that age stamp. As a result, it is easy to separate the heap or stack data contributing to the attack from stale tainted data left by a previous function frame.

Besides age stamps, this tagging method tracks a small amount of additional meta-data. For instance, it inserts red markers just above and below a buffer allocated on the heap. An overflow of this buffer triggers a reaction in the emulator (e.g., to log the buffer contents for later correlation). In addition, we maintain two bits per tainted byte to distinguish between different overlapping tainted buffers with the same age stamp.

The details and analysis of age stamps and related meta-data is quite complex and beyond the scope of this paper. Interested readers are referred to [23]. What is important for this paper is that when a buffer overflow attack is detected, the origin pointers, age stamps and additional meta-data combined allow us to determine with great accuracy the exact bytes that contributed to an overflow. The overhead of age stamps in memory consists of an additional word per tainted address. The computational overhead is modest, less than 20% compared to single-origin tracking without age stamps in real applications like Apache.

In principle, any tagging method can be used for *Hassle*. However, as alignment is error prone in black-white tagging, we mostly used single-origin tracking for our experiments, and age-stamp tracking where indicated.

2.2 Retainting

Regardless of tagging method, origin pointers are useless in the case of encrypted channels. The reason is that without the key we cannot perform the one-to-one mapping between bytes in memory and bytes in the network trace.

For this reason, *Hassle retaints* all encrypted data. Rather than pointing to a specific byte in the encrypted network trace, we make it point to a

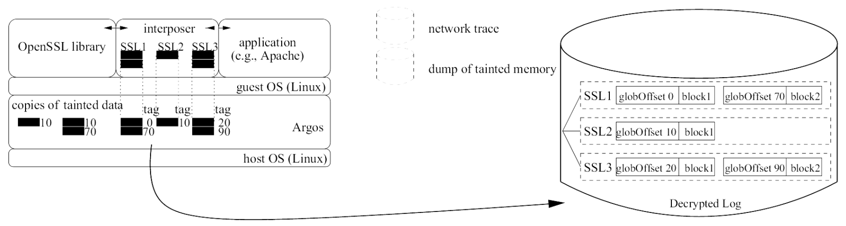

Fig. 2: Global Offsets in the Decrypted Data Log.

specific byte in the *decrypted* SSL stream. Of course, the nature of decrypted streams is different from that of the network trace. For instance, layer 2-4 headers are not visible and TCP flows have already been reassembled. As a result, we will have to adjust the signature generation and filtering components accordingly.

Two implementation issues remain. First, after separating encrypted and non-encrypted data we must retaint the data right after decryption in such a way that a tag used for retainting is unique across all streams. As a result, the tag cannot be a simple offset into any one particular SSL stream. Second, we should be able to uniquely identify SSL conversations and associate incoming data with an SSL stream. In the next two sections, we discuss our solution to each problem separately.

2.2.1 Determining the tag

As decryption occurs in user space, we employ light-weight interposing libraries between the application and the SSL functions. Whenever a read is performed on an SSL stream, the data will be decrypted. At that point the interposer requests a retaint for the decrypted data and logs the decrypted data to file. Beyond that, the interposer serves as a low-overhead relay between the SSL library and the application.

While the interposer trivially knows the offset of decrypted data in the corresponding SSL stream, determination of the tag should not take place there. Given a tainted data item, *Hassle* must be able to identify exactly the decrypted SSL block in which it originates. In other words, a tag must be unique not only within its own SSL stream, but across all streams. Doing such retainting in the interposer requires adding a unique SSL stream identifier to each tag, which is both complex and expensive in memory.

Instead, we perform trivial retainting in the emulator and push all complexity to detection time. For the remainder of this section, refer to Figure 2 which zooms in on the decrypted data log and shows a situation where three SSL connections are active; the decrypted data blocks in the channels are tagged by the emulator.

We maintain, conceptually, a single log for all SSL streams and let the emulator determine a tag consisting of an offset in this global log. In reality, we store each SSL stream in separate append-only logs identified by a unique SSL stream identifier (the nature of which will be discussed in Section 2.2.2). For instance, the tags in Figure 2 refer to an offset in the global input. That is, the blocks that are tagged 0, 10, and 20 indicate that the first block starts at global offset 0, and since the next block starts at offset 10, the first block contains 10 bytes. Similarly, the third block starts at offset 20, so the second block also contains 10 bytes. However, while this is the second block in the global input, it is the first block in SSL stream 2. For completeness, the figure also shows on the left some tainted data that has been copied, leading to tag propagation.

In other words, *Hassle* orders and tracks all updates to the decrypted data log in a global order, layering a virtual append-only global log over the individual SSL stream logs. The log for SSL stream 1 in Figure 2 contains two data blocks, containing 10 and 20 bytes respectively. The global log, on the other hand, consists of five data blocks. Blocks 1 and 2 both contain 10 bytes; block 3 contains 50, and so on. *Hassle* tags the decrypted data with an offset into the global log, trivially guaranteeing uniqueness. When an attack is detected, the tags of offending bytes point to a specific block in the global log. We maintain a simple index to find the corresponding SSL stream and hence all decrypted data.

Finally, for each SSL stream we also store the original tag of the first decrypted data block. This tag points to a byte in the encrypted network trace where it originated. Thus, we are always able to find the network flow that carried the attack, which in turn enables us to identify the IP addresses and port numbers of the attack.

In summary, for retainting the interposer asks the emulator to determine a new tag for the data as explained earlier. It then pushes the decrypted data to the decrypted data log. Currently, this is implemented as a request over a UDP connection to the host OS. As UDP is unreliable, we take into account potential reordering and loss. For the first chunk of decrypted data in the SSL stream, we also log the association between the decrypted data and the original tag, enabling us to recover conveniently the network flow (IP addresses, ports, etc.) in which an attack originated.

The administration of all other meta-data works in exactly the same fashion as in the non-encrypted version. In particular, this is true for the meta-data that is kept for age stamp tracking, such as age stamps and red markers as described in Section 2.1.

2.2.2 Identifying the SSL conversation

The construction of a unique identifier for a single conversation is not trivial. Ideally, the identifier should be a unique number derived from one or more fields of the SSL connection structure. Simply using the memory address of the `ssl` structure (see Listing 1) will not suffice, because new conversations may reuse the structures associated with old conversations.

However, the handshake phase of SSL (version 3) connections includes the exchange of unique challenges by client and server, which can be obtained from the SSL structure. Unlike client challenges, server challenges cannot be influenced by clients, and are thus well-suited for identifying the conversation. Unfortunately, SSL version 2 does not support server challenges. While older versions of SSL are not our main concern, we decided to add some support for version 2. For such conversations, we currently resort to a combination of the client challenge with the memory address of the SSL connection structure and the thread id of the process using the OpenSSL library. Admittedly a hack, the values of the latter two are not controllable by any attacker and the combination is pseudo-unique.

2.3 Interposition details

SSL conversations start with a handshake phase that deals with authentication and creation of a session key. No application data is transmitted during this phase and we therefore do not monitor it. This phase also creates the SSL structure for the conversation.

Whenever an application calls `SSL_read` to decrypt and read data, we intercept the call by way of library interposition. Besides the call to `SSL_read`, we are interested in a small subset of other calls, including `SSL_shutdown` and `CRYPTO_num_locks` and a few others[1]. As an example, we show the code for the `SSL_read` interposer in Listing 1. In the first few lines we find (line 2) and execute (line 3) the real `SSL_read` function as requested by the client. Next, we retaint the data and log the decrypted stream using the `retaint_netidx` and `inform_logclient` functions, respectively. None of the retaint functions are visible to the caller, rendering the interposer transparent to the client.

[1] In fact, we interpose SSL_write also, but the reasons for doing so are related to attack replaying and beyond the scope of this paper.

Listing 1: Interposer for `ssl_read` library function

```
int SSL_read(SSL *ssl, void *buffer, int length) {
  int(*func)() = (int(*)())dlsym(RTLD_NEXT,"SSL_read");
  int func_result = func(ssl, buffer, length);
  retaint_netidx(...); // now retaint
  inform_log(...);     // log decrypted data (Fig. 1)
  return func_result;  // return original result
}
```

A call to `SSL_shutdown` simply leads to destruction of state maintained by *Hassle*. `CRYPTO_num_locks` is more complex. OpenSSL uses a number of global data structures that will be implicitly shared when multiple threads use the library. To use the library in the context of threads safely, we need locks to prevent simultaneous access to the global structures and `CRYPTO_num_locks` returns the number of locks needed by the library to synchronise access. Because our interposing library also implements a global data structure, the number of locks should be increased by one. So we interpose this function to make another lock available to protect the global data structure.

3 Signature generation

As illustrated in Figure 1, signature generation is devolved from detection and different generators can be plugged into the architecture. We currently support two main classes of generator that will be referred to as *pattern-based* and *vulnerability-based*, respectively. Pattern-based signatures are widely used in network intrusion detection systems such as Snort [21] and Bro [15]. They consist of a basic identification of the type of packet (e.g., TCP or UDP and port number), together with a byte pattern which is matched against traffic of the appropriate type.

In vulnerability-based signatures we focus on buffer overflows on the heap and stack and decouple the signature from the attack's content in bytes completely. The signature consist of a bound on the combined length of a set of protocol fields. Any message where such fields have a combined length that exceeds this bound will incur an overflow, regardless of their content, so these signatures cater well to polymorphic attacks. On the surface, they are quite similar to those of Covers [10], but we will show that they are considerably more accurate.

Each class of signatures has four variants of generators: (1) encrypted vs. non-encrypted, and (2) single-origin net tracking with age stamps vs. single-origin net tracking without age-stamps. The main difference between encrypted and non-encrypted channels, as far as signatures are concerned, is where they are applied. For non-encrypted channels, we are able to apply signatures in the network before the malicious traffic reaches the host (indicated by (5) in Figure 1). In contrast, encrypted channels require the filters to be applied at a higher level, i.e., as an *interposer filter* in user-space (indicated by (6)). In addition, an interposer filter must know which signatures to apply. To do so, the signature generators consults the network tag that was stored in the decrypted data log to find the corresponding flow in the network trace. By means of the flow, we obtain the port numbers used in the attack (and possibly other network-specific information). Finally, the forensics data generated by *Hassle* specifies details about the application under attack. This is then used by remote clients to determine which interposer filters should apply the filter.

3.1 Pattern-based signatures

Our pattern-based signatures handle all buffer overflows and all code injection attacks (with slightly better signatures for buffer overflows). Note that while code injection may be caused by buffer overflows, there also exists exploits for double frees, format strings, etc. Regardless of exploit, *Hassle* is able to fingerprint such attacks also. We distinguish between signature generators with and without age-stamp analysis.

*Single origin net tracking without age stamp analysis (***SontNoAsa***).* Whenever *Hassle* detects an attack, we determine whether the program counter (EIP) register was tainted. If so, we use the origin pointer of the register to locate the corresponding byte in the network trace. Next, we perform a correspondence search between the flow content in the network trace prior to this byte and the tainted data in memory. Whenever the tags point to the appropriate values in the trace, we include them in the pattern. Pseudo-code for this naïve algorithm is shown in Listing 2.

Listing 2: Naïve generator for pattern-based signatures

```
char *addr = top of memory location loaded in EIP;
sig_t sig = { *addr }; // signature as byte sequence
while (tag(addr-1) == tag(addr)-1)
    sig = concatenate (*--addr, sig );
```

Hassle improves on this naive scheme by taking into account simple gaps in the tainted memory region and Unicode character encodings (e.g., UTF-8). Gaps may occur in tainted buffers for many reasons, e.g., due to non-tainted assignments to overflown memory *after* the buffer overflow occurred and *before* the control flow was diverted. For instance, consider the (contrived) code snippet in Listing 3.

Listing 3: Tainted data: gaps in tainted data

```
01 void read_from_socket (int fd) {
02   int n;
03   char unrelated_1 [8];     // not vulnerable
04   char vuln_buf [8];        // vulnerable buffer
05   char unrelated_2 [8];     // not vulnerable
06   read (vuln_buf, fd, 32);  // overflow
07   read (unrelated_1, fd, 8);// tainted gap
08   read (unrelated_2, fd, 8);// adjacent buffer tainted
09   n = 1;                    // untaints 4 bytes: gap
10   return;
11 }
```

While the code is not very realistic, it serves to illustrate a number of complications that prevent the naive solution from producing correct results in *some* cases. Before the attack is detected (when `return` is executed), the assignment in line (9) creates an untainted gap of 4 bytes in the tainted buffer. Similarly, the read in line (7) creates a tainted gap filled with unrelated data. Finally, the vulnerable buffer may adjoin another buffer that also contains tainted data, as demonstrated by the read in line (8).

In the pattern-based signatures generated using SontNoAsa, gaps are detected by looking at the tags. Gaps either have no tags, or tags with unexpected values. Gaps are skipped whenever the byte on the other side of the gap has the appropriate (expected) tag value. In case there is no such byte, the signature stops here. Unfortunately, the adjacent buffer `unrelated_2` that also contains tainted data cannot be distinguished from the vulnerable buffer and is therefore also included in the signature. As a result, SontNoAsa and pattern-based signature can lead to false negatives.

Hassle can easily cater to well-known forms of encoding like Unicode. Sometimes a network trace carries ASCII data, which is translated to a Unicode representation in memory, or *vice versa.* Either way, the skew between network trace and memory is predictable. As both gap- and Unicode handling are trivial extensions to the naive algorithm in Listing 2 we will not show them here.

*Single-origin net tracking with age stamp analysis (***SontAsa***).* Applying age-stamp analysis improves the accuracy of pattern-based signatures in the case of buffer overflows attacks on heap or stack. Extended age-stamp analysis directly yields all bytes in memory that were used in the overflow. By means of single-origin net tracking we obtain the corresponding network bytes. As we only identify the relevant bytes, compensating for gaps and Unicode is no longer necessary, as it is handled automatically. In fact, the signature generation is considerably more accurate, as age stamp analysis is also capable of distinguishing buffers `vuln_buf` and `unrelated_2`[2].

Code injection signatures. Not all attacks are overflows. Perhaps other means were used to divert control to the code injected by the attacker. Since such code is tainted by nature, the attack is detected when instructions in the tainted region are executed. On rare occasions, injected code may be executed because of legitimate control flow (i.e., a *bona fide* jump to a memory area that is tainted). More commonly, the jump is the result of a control flow diversion by means of a *format string* attack, *heap corruption*, or the *overflows* mentioned earlier. Regardless of how the injected code is reached, we again align memory and network trace to generate a signature. The only difference is that if the attack is not an overflow, we match against the injected code.

In case we detect both a buffer overflow and code injection (i.e., the buffer overflow was used to divert control to the injected code), we have to decide whether to use as signature either the match against the injected code, or the match against the overflow bytes. As injected code is more likely to be polymorphic than the exploit itself, we favour the overflow signature. As a rule of thumb we use the injected code signature only if (a) it is longer than the overflow signature, and (b) if the length of the overflow signature is less than some minimum length L_{min} (e.g., L_{min}= = 12).

Limitations. Pattern-based signatures are attractive because of their simplicity, and their popularity in existing IDSs. Unfortunately, they are also fairly weak and incur both false positives and false negatives. In particular, by using the actual content of the attack, traffic pattern-based signatures are powerless against polymorphic attacks. They also do not work when multiple tainted buffers are adjacent in memory.

3.2 Signatures for polymorphic buffer overflows

For polymorphic buffer overflow attacks we decouple the signature from an attack's content in bytes. Instead, we look at the *vulnerabilities*. A message that causes a buffer overflow contains one or more protocol fields of unusual length that, when copied collectively into a vulnerable buffer, overwrite critical data. Vulnerability-based signatures establish a maximum length *L* for the field(s). Any message where the combined length of these fields exceeds *L* is sure to overflow the buffer. We first discuss a naive implementation that is also used by other projects and demonstrate why

[2] And indeed more complicated cases. For details, see [23].

it is flawed. Next, we explain how age stamp analysis helps us solve the problems.

*Single-origin net tracking without age-stamps (***SontNoAsa***)*. In this naive implementation, we trace the point of attack *X* to a byte *N* in the network trace and establish what protocol field *P* contains this byte. Doing so is trivial if traffic is not encrypted. In our case, we reassemble the TCP stream and dissect the higher-layer protocols with a protocol dissector (we use a modified version of Ethereal [2]). After locating the protocol field containing *N* we generate a signature consisting of an identification of the stream and application (port numbers, executable name) together with a bound *L* on the length *P*, where *L* is (*N* - start of *P*). It is likely that any message with *P* longer or equal to *L* results in an overflow (but not certain, as we shall see shortly).

For encrypted traffic the procedure is a little more complicated as we cannot start from the network packets to dissect the input stream. As we start dissecting above the transport layer, how do we decide which protocol dissector to use? We identify three solutions for dealing with the problem. First, we may use custom interposers for specific applications. For instance, we can apply an HTTP interposer for Apache which always assumes HTTP traffic. Second, we may use the port numbers in the network trace as an indication of the protocol (e.g., all port80 traffic will be assumed to be HTTP). Third, we may use the information about the application as an indicator for the protocol (if the application is “apache”, we use the HTTP dissector). Currently, we use hard-coded associations.

In our implementation, we modified the Ethereal protocol analyser [2] to start from higher-level protocols and to work with incomplete protocol messages. As this signature generator is very similar to Covers [10], we refer to it as *Hassle*-Covers.

Limitations. Unfortunately, Covers (and thus *Hassle*-Covers) yields both false positives and false negatives. First, exploits like Apache-Knacker [22]) use the fact that sometimes multiple protocol fields are copied in the same buffer to generate an overflow of the buffer with the content from all these fields. As a result, establishing a bound on the length of a single field may miss attacks where the length of *P* is small, but the combined length of all fields exceeds the length the buffer. Similarly, it may misdiagnose a message as malicious when *P* is longer than *L*, even though the combined length of all the relevant fields is less than the buffer size.

The second reason is related, but more subtle. It is also more serious. The dissector used to generate signatures may work at different protocol field granularities than the application itself. For instance, the dissector may identify subfields in a record-like protocol field as separate fields, while the application simply treats it a single protocol field. As a consequence, the two types of misclassification described above may occur even if the exploit does not explicitly use multiple fields. As we generally do not have the application's source code, and hence have no knowledge about the granularity of the application's dissector, this is a serious problem.

*Single-origin net tracking with age-stamp analysis (***SontAsa***)*. To deal with this problem we take into account *all* bytes used in the overflow. A reliable way of establishing which bytes were used in the exploit is by means of age-stamped net tracking. In the case of non-encrypted traffic we find those bytes in the network trace directly. In the case of encrypted traffic those bytes are found in the decrypted data log using the modified Ethereal protocol dissector, as described in the previous sections.

To be precise, we find accurately *all* bytes that were used in the overflow and we do so before a single instruction of the attack is executed and without the need to replay the attack. Gaps in the buffer overflow (as explained in Section 3.1), be they tainted or non-tainted, are duly skipped, and adjacent buffers that are both tainted but different are separated. In addition, encodings like Unicode are automatically handled. The details are complex and beyond the scope of this paper. The exact procedure is explained in [23].

Given the overflow bytes, we then identify *all* protocol fields that were used in the attack and establish an upperbound L on their combined length *according to our dissector*. Whether or not the application uses a different protocol field granularity is now immaterial.

4 Filters

Hassle filters for non-encrypted traffic consist of simple checks, either matching pattern-based signatures against network packets, or looking at the length of fields of specific protocol messages for vulnerability-based signatures. They can be applied in the network or in the operating system kernel.

For encrypted channels, similar procedures are used, except that they are applied by means of library interposition in user-space. Filters should only apply those signatures that apply to the specific application that uses the SSL library. Currently, this is done by explicitly associating a separate interposer filter library with every application we want to protect. Each interposer filter only picks up the signatures for the application it is protecting. Since *Hassle* provides the full name of the executable as part of the signatures, the association is trivial. For vulnerability-based signatures, the interposer filters again use the modified version of Ethereal for protocol dissection.

5 Results

For realistic performance measurements we compare the speed of code running on *Hassle* with that of code running without emulation. While this is an honest way of showing the slowdown incurred by our system, it is not necessarily the most relevant measure, as we use *Hassle* as a honeypot rather than a desktop machine. To our knowledge, no automated attacks exist that shun slow hosts, because they might be honeypots.

It should also be mentioned that encryption is known to be one of the most challenging applications for dynamic taint analysis, because decryption requires a large number of tainted operations. For instance, recent work on demand emulation [6] describes a technique to speed up emulation-based taint analysis by switching to fast VM-mode when possible. While many applications incurred as little as a factor 2 slowdown, SSL incurred a slowdown of 150.

Performance. To quantify the observed slowdown we used the Apache 2.2.3 web server using the OpenSSL library. The first simple test consisted of requests to read a 5MB block from the client to the server, which on top of a vanilla Qemu emulator took Apache 19.9s to complete (2.06Mbps). On *Hassle*, the same task took 23.47s (1.75Mbps), incurring a 15% overhead.

We also evaluated Apache throughput in terms of number of processed requests per second and the corresponding average response time. We used `httperf`[3] for generating requests. The experiments were conducted on a dual Intel™ Xeon at 2.80GHz with 2MB of L2 cache and 4GB of RAM. The system was running SlackWare Linux 10.2 with kernel 2.6.15.4.

The results for https (using SSL) are summarised in Table 1. The table lists results for 3 Apache configurations: (i) running natively, (ii) running on the Argos honeypot, and (iii) running on *Hassle*. We also show some results for non-encrypted http connections for comparison[4]. The results are the best possible in the sense that at this rate the webserver was able to keep up fully with the request rate, while not yet incurring unreasonably long response times. For instance, for all reported rates the response times

[3] www.hpl.hp.com/research/linux/httperf/

[4] We were unable to measure reliably the native version for plain http, because httperf at the client side became the bottleneck.

were below 200ms. Beyond these rates, response times shot up to many hundreds or even thousands of milliseconds.

description	average (req/s)	standard deviation	relative to native	response time (ms)
https/native	57.0	0.3	1.0	21
https/Argos	0.6	0.07	95.0	87
https/Hassle	0.55	0.12	103.6	63
http/Argos	38	1.8	n/a	147
http/Hassle	38	1.7	n/a	200

Table 1: Maximum rates for https connections.

The experiments confirm that SSL is very expensive for dynamic taint analysis, incurring a slowdown of approximately a factor 100 over native code running SSL, and a factor 70 over non-encrypted channels using the same (emulated) configuration. Consequently, dynamic taint analysis for SSL encrypted channels is only viable on honeypots, and even here the number of connections should be limited. Note however, that slowness is not really a major issue for a honeypot as long as it is able to serve a request sufficiently fast. Moreover, in most deployments of honeypots like Argos (e.g., at SURFnet[5], Eurecom [9], and in the Noah project[6]), a first-pass filter of low-interaction honeypots is used to shield the high-interaction honeypot from most requests. The second thing to observe is that there is little difference between *Hassle* and the original Argos (i.e., a honeypot without retainting and logging of encrypted data).

Probing further, it appeared that most of the overhead is in the connection set-up where SSL uses asymmetric encryption. As a result, performance improves significantly when use is made of https sessions. For instance, for 100 sessions per connection, the reply rates for https *Hassle* are shown in Table 2.

description	average rate (req/s)	relative to native
https/native with sessions	486	1.0
https/Argos with sessions	31.1	15.6
https/Hassle with sessions	25.0	19.4

Table 2: Maximum rates for https connections with sessions

Micro-benchmarks. Retainting itself is not very expensive. We measured 200µs on a Pentium M at 1.4GHz with 1GB of RAM, running Ub-

[5] http://honey.surfnet.nl/

[6] http://www.fp6-noah.org/

untu Linux 6.0.6. The guest OS ran Ubuntu Linux 5.05 with kernel 2.6.12.9, on top of Qemu 0.8, Argos and *Hassle*. Similarly, the overhead of the entire interposition library to do the retainting is modest. We measured performance with and without the interposition library for both SSL reads and SSL writes for block sizes ranging from 100B to 16kB bytes. For writes, the relative overhead lies between 17.5% for the largest blocks and 26% for the smallest ones. For reads, the results range from 24.5% for the largest to 50% for the smallest blocks. Likewise, in our evaluation the interposer filters that scan SSL streams for the occurrence of a signature incurred overheads between 2% and 15% compared to a system without the filters, for the most expensive (pattern-based) signatures.

Generating a single-origin net tracking signature. In a cursory evaluation of the signature generator we tested the generators for pattern-based and vulnerability-based signatures using request sizes of 170B and 100kB, respectively (representing different amounts of data to dissect and/or scan). The pattern-based signature was generated in 1.1ms for the small request, and 14ms for the large one (median values). Vulnerability based signatures for polymorphic attacks required 3.9ms, and 26ms to be completed (assuming the protocol dissector is loaded already).

6 Related work

To our knowledge, we are the first to tackle the problem of one-shot signature generation for communication on encrypted channels. Dynamic taint analysis, on the other hand, is well-known and used in TaintCheck [14], Vigilante [12], and Argos [18]. None of these projects offer signatures for encrypted traffic.

Library interposition as a way of monitoring interaction with libraries is used frequently to analyse applications [3] and generate audit trails [8]. Liang et al. propose library interposition to learn about program inputs that lead to crashes induced by buffer overflows [11]. In essence, they consider library calls made from a given program context and raise an alert when an input is significantly longer than the maximum input length seen in the past. Interposition is also applied at the system-call level either to confine the application [5], or to monitor the compliance of a sequence of calls with a predefined application model [4, 11]. In contrast, we intercept library calls to switch to tracking decrypted network streams by adjusting the tags in dynamic taint analysis.

Several of our signature generators are based on existing work. In particular, the pattern-based signatures are quite popular in open-source

NIDSs like Snort [21] and Bro [15]. However, the way we generate the signatures is a little different from existing projects. This is true for the way *Hassle* skips gaps and handles Unicode, and even more so for the age-stamped net tracker that determines accurately which bytes are used in a buffer overflow.

Similarly, the single-field vulnerability-based signature is already proposed in Covers [10]. We have demonstrated that such signatures have fundamental flaws and shown how we solved them.

Application-level filtering is performed by virus scanners and Vigilante. Filters in interposing libraries are not very common. While the paper is a bit vague about it, we suspect that they are also used in ARBOR [11], although the filters are of a very different nature.

7 Conclusions

We have described *Hassle*, a honeypot system that is capable of generating signatures for communication over both encrypted and non-encrypted channels. For encrypted traffic we retaint the tainted data by making the tags point to the decrypted SSL streams. Different types of signature generator can be used in the system. Which one should be used is a tradeoff between simplicity and accuracy. In our opinion, pattern-based signatures are useful for simple, non-polymorphic attacks, while vulnerability-based signatures work well with more advanced, polymorphic exploits. To our knowledge, we are the first to develop a system capable of fingerprinting attacks over encrypted channels and cater to both monomorphic and polymorphic exploits.

8. References

[1] P. Bueno. IIS Exploit released / Gagobot.XZ - SANS Microsoft Advisories. http://isc.sans.org/diary.html?date=2004-04-14, April 2004.
[2] G. Combs. Ethereal network protocol analyzer. http://www.ethereal.com.
[3] T. W. Curry. Profiling and tracing dynamic library usage via interposition. In *Usenix ATC*, Boston, MA, June 1994.
[4] J. Giffin, S. Jha, and B. Miller. Efficient context-sensitive intrusion detection. In *11th Annual Network and Distributed Systems Security Symposium*, 2004.
[5] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 6th Usenix Security Symposium*, 1996.

[6] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. *SIGOPS Oper. Syst. Rev. (Proc. of ACM SIGOPS EuroSys, April 2006, Leuven, Belgium)*, 40(4):29–41, 2006.

[7] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. *8th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Sept 2005.

[8] B. A. Kuperman and E. Spafford. Generation of Application Level Data via Library Interposition. Technical Report CERIAS TR 1999-11, 1999.

[9] C. Leita, M. Dacier, and G. Wicherski. SGNET: a distributed infrastructure to handle zero-day exploits. Technical Report EURECOM+2164, 2007.

[10] Z. Liang and R. Sekar. Fast and automated generation of attack signatures: a basis for building self-protecting servers. *CCS '05*.

[11] Z. Liang, R. Sekar, and D. C. DuVarney. Automatic synthesis of filters to discard buffer overflow attacks: A step towards realizing self-healing systems. In *USENIX Annual Technical Conference - short paper*, Anaheim, CA, 2005.

[12] M. Costa, J. Crowcroft, M. Castro, A Rowstron, L. Zhou, L. Zhang and P. Barham. Vigilante: End-to-end containment of internet worms. In *Proc. of the 20th ACM Symposium on Operating Systems Principles*, Brighton, UK, 2005.

[13] McAfee. Encrypted Threat Protection - Network IPS for SSL Encrypted Traffic. white paper, February 2005.

[14] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2005.

[15] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31:23–24, December 1998.

[16] F. Perriot and P. Szor. An analysis of the slapper worm exploit - white paper. Technical report, Symantec Security Response, 2002.

[17] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Network-level polymorphic shellcode detection using emulation. In R. Büschkes and P. Laskov, editors, *DIMVA*, volume 4064 of *Lecture Notes in Computer Science*.

[18] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks. In *Proc. ACM SIGOPS EUROSYS'2006.*

[19] C. Prosise and S. U. Shah. Hackers' tricks to avoid detection. WindowSecurity White Paper, http://secinf.net/info/misc/tricks.html, 2002.

[20] N. Provos. Improving host security with system call policies. In *12th USENIX Security Symposium*, Washington, 2003.

[21] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the 1999 USENIX LISA Systems Adminstration Conference*.

[22] SecurityFocus. Can-2003-0245 apache apr-psprintf memory corruption vulnerability. http://www.securityfocus.com/bid/7723/discussion/, 2003.

[23] A. Slowinska and H. Bos. Prospector: Accurate analysis of heap and stack overflows by means of agestamps. Technical Report IR-CS-031, Vrije Universiteit Amsterdam, June 2007.

[24] T. Toth and C. Kruegel. Accurate buffer overflow detection via abstract payload execution. In *Recent Advances in Intrusion Detection*, 2002.