Prospector: a protocol-specific detector of polymorphic buffer overflows

Asia Slowinska, Georgios Portokalidis, and Herbert Bos
Vrije Universiteit Amsterdam
{asia,porto,herbertb}@few.vu.nl

Abstract

While future attacks are likely to be increasingly polymorphic, current intrusion detection methods tend to be either powerless in the face of attack mutation, or too inaccurate to be used as intrusion prevention filters. Our system, known as *Prospector* consists of three components. First, an emulator-based honey-pot uses taint analysis to detect zero-day intrusion attempts with great accuracy. Second, a signature generator uses protocol knowledge to generate signatures for use in filters. Third, a filter engine functions as an intrusion prevention system, scanning incoming traffic for many different signatures simultaneously in an efficient manner and dropping streams containing attacks. No false positives are incurred on either the detection or the filtering phase. The *Prospector* IPS is implemented both on a general purpose CPU and on a network processor embedded on a network card.

1 Introduction

During an informal discussion at the 17th TF-CSIRT meeting in Amsterdam several CERTs complained about the lack of defense against automated polymorphic attacks. In the future polymorphism is expected to be common and the need for counter-measures is urgent. Not surprisingly, research on automatic signature generation for such attacks has become a hot topic [3,5–8,15]. Most of the proposed solutions target detection rather than prevention and incur unacceptable false positives (FP) ratios to be used in filters. However, with worm propagation rates beyond human response times, prevention and containment are crucial [4,14].

In this paper we discuss *Prospector*, a protocol-specific detector of polymorphic buffer-overflow attacks to be used as an IPS. *Prospector* consists of three components. First, an emulator-based honey-pot uses taint analysis to detect zero-day intrusion attempts with great accuracy. Second, a signature generator uses protocol knowledge to generate signatures for use in filters. Third, a filter engine functions as an IPS, scanning incoming traffic for many different signatures simultaneously in an efficient manner and dropping streams containing attacks. While we refrain from claiming a *guaranteed* FP ratio of zero, in practice, we found no FPs for either detection or filtering and for now we do not see how they could occur in practice.

Prospector is founded on the same principles as COVERS which uses knowledge about the protocols being used to trace buffer overflows to protocol fields [8]. In this approach a host-based IDS pinpoints the address that caused a control flow diversion as a result of a buffer overflow. Next, the address that causes the alert is matched with a a value in a specific protocol field by the protocol-aware signature generator. The signature generator generates a signature, for instance by determining the maximum length L for the protocol field. This value can then be used to check all traffic that corresponds to this protocol. If the length of the protocol field exceeds L the message will also cause an overflow, regardless of its actual contents. In other words, we focus on vulnerabilities rather than exploits.

The contributions of this paper are as follows. First, we improved the protocol-specific method to make it more accurate, incurring fewer false positives (FPs) and false negatives (FNs). Second, *Prospector* is based on different detection methods and arguably implements a more mature IPS. Third, we have experimented with different implementations of *Prospector* to try out different application scenarios. Concretely, *Prospector* IPS is implemented both on a general purpose CPU (for host-based protection) and on a network processor embedded on a network card such as might be used by a line card in a router, close to the edge.

The remainder of this paper is organised as follows. Section 2 discusses related work. In Section 3 we describe our architecture which we evaluate in Section 4. We conclude in Section 5.

2 Related Work

A crude approach to detect polymorphic attacks is by looking at flow aggregates, e.g., detection techniques that trigger alerts for unusual numbers of connections to unique IP addresses [15], or anomalies in webtraffic [12].

Hamsa's signature generator is based on the invariant bytes in attacks. It is fast and offers bounded FPs and FNs [7]. Polygraph, which precedes Hamsa, is also based on invariant byte strings common to different mutations of a worm [5]. As specific contiguous byte sequences, such as protocol framing strings and the high order bytes of buffer overflow return addresses, normally remain constant across instances of a polymorphic worm, these can be used to generate a worm signature. As we aim for use in IPS, our goal differs from that of Hamsa and PolyGraph in that we try to reduce the number of FPs to zero.

Prospector centers on what may be termed vulnerability-based signatures. Compared to the vulnerability-based system in [3], our signatures are simpler, without requiring extensive preprocessing of the applications to generate/use the signatures.

The problem of polymorphic attacks is also addressed in [6] which employs structural analysis of binary code to identify structural similarities between different worm mutations. Again, the goal of [6] differs from our approach in that it targets detection rather than prevention. Phrased differently, they do not need to reduce the FP ratio to zero. Also, it seems that structural analysis is difficult if the exploited protocol fields are encoded (e.g., URL encoding).

Vigilante offers zero FPs and limited protection against polymorphism [9], but it is fairly easy to generate polymorphism beyond its capabilities. Also, the signature generator relies on replaying attacks, which may be very hard (especially in the face of challenge/response authentication).

A different approach is to exploit knowledge about the protocol fields. Like Covers [8], we trace the address that causes the control flow diversion to a specific (higher-level) protocol field and capturing characteristics (such as the length of the field) that are subsequently used as a signature. Covers uses address space randomisation (ASR) to detect an attack. Any attempts to divert the control flow will, with high probability, crash the process with a memory fault. If so, the OS is queried to find the address M_t that caused the crash (Figure 1). Next, the (logged) traffic is scanned for the address A and some bytes in its vicinity, thus approximating N_t. Using knowledge about the protocol, Covers determines the protocol field that caused an overflow, and uses the length of this field as a signature, as all messages with the same field with this length lead to the same overflow. By employing properties like field length, signatures are independent of actual content and hence resilient to polymorphism.

Prospector builds on the same principles, but differs in important aspects. First, rather than the somewhat error-prone ASR, we use taint analysis for detecting intrusions. With ASR there is a non-negligible chance that the attack does not cause a memory fault immediately, but crashes after executing a few random instructions which would render the address useless. Second, by tracking the origins of tainted data in memory in the network trace the correlation between memory and network trace can be exact. In our experience the probability of making the wrong guess as to the origins N_t of the address A that overflows M_t in the network trace is high [11]. Worse, if protocol fields are encoded in the network trace (e.g., URL encoding), scanning for occurrences of the target fails altogether.

Third, Covers is unable to stop sophisticated overflows which are caused by more than one protocol field (e.g., Apache-Knacker [13]). Consider, for instance, chunking and multiple host headers in HTTP[1], where multiple chunks or headers end up in the same buffer. Such attacks may lead to FPs in Covers, but are correctly identified by *Prospector*.

Fourth, while Covers uses L_1, the length of the entire protocol field, *Prospector* is more precise and considers only distance L_2, from the start of the protocol field to the point where the control flow diversion occurred. Using L_1 may cause FNs, as a signature generated for a long version of the protocol field will not find attacks with shorter fields, even if they contain the exploit. We speculate that Covers takes the entire field because it is unable to pinpoint N_t accurately, as jump targets are often repeated in the exploit to handle differences in offset (as indicated by multiple occurrences of A in Figure 1).

Fifth, the way multiple signatures are used in [8] is not specified. We have developed an efficient tree-like structure for dealing with large numbers of signatures as explained in Section 3.4.

Sixth, *Prospector* has an option to scan for and reject malformed protocol messages (protocol scrubbing).

3 Architecture

Prospector protects hosts from buffer overflow attacks by tracing the address that causes the control flow diversion to a specific (higher-level) protocol field and capturing characteristics (such as the length of the field) that are subsequently incorporated in an attack signature. The signature consists of what we call *critical* field

[1]Throughout this paper, we use HTTP as a running example, but we stress that *Prospector* caters to any protocol.

properties (e.g., a maximum length for the URL field) combined with *value* fields that characterise the context (e.g., that the URL field should be checked in HTTP GET messages).

3.1 Attack Detection

For attack detection we use *Argos*, an efficient and reliable emulator that tags and tracks untrusted network data and triggers an alert whenever the use of such data violates security policies [11]. Tagging and tracking network data is commonly known as taint analysis [10]. In other words, data originating from the network is marked as tainted, whenever it is copied to memory or registers, the new location is tainted also, and whenever it is used, say, as a jump target, we raise an alarm.

Argos extends the *Qemu* [1] emulator by providing it with the means to taint and track memory and registers, and to generate memory footprints in case of a detected violation. *Qemu* translates all guest instructions to host native instructions by dynamically linking blocks of functions that implement the corresponding operations. Tracking tainted data involves instrumenting these functions to manipulate the tags, as data are moved around or altered. In summary, all network traffic is tagged and tainted.

Most of the observed attacks today gain control over a host by redirecting control to instructions supplied by the attacker (e.g., shellcode), or to already available code by carefully manipulating arguments (return to `libc`). For these attacks to succeed the instruction pointer of the host must be loaded with a value supplied by the attacker. In the `x86` architecture, the instruction pointer register `EIP` is loaded by the following instructions: `call`, `ret` and `jmp`.

By instrumenting these instructions to make sure that a tainted value is not loaded in `EIP`, we identify all attacks employing such methods. While these measures capture a broad category of exploits, they alone are not sufficient. For instance, they are unable to deal with format string vulnerabilities, which allow an attacker to overwrite any memory location with arbitrary data. These attacks do not directly overwrite critical values with network data, and might remain undetected. Therefore, we have extended dynamic taint analysis to also scan for code-injection attacks that would not be captured otherwise. This is accomplished by checking that the memory location pointed to by `EIP` is not tainted. Finally, to address attacks that are based solely on altering arguments of critical functions like system calls, we may check system call arguments.

When a violation is detected, an alarm is raised and the signature generation phase starts. To aid signature generation, *Argos* dumps all tainted blocks to file, with markers specifying the address that triggered the violation, the memory area it was pointing to, etc.

In addition, we employ a novel technique to automate forensics on the code under attack. When an attack is detected, *Argos* does not yet know which process is causing the alarm. To unearth additional information about the application (e.g., process identifier, executable name, etc.), we inject our own shellcode to perform forensics. In other words, we 'exploit' the code under attack with our own shellcode.

The dump of the memory blocks (tainted data, registers, etc.) plus the additional information obtained by our shellcode is then used for correlation with the network traces in the trace database.

3.2 Information Correlation

This step correlates the address causing the violation with the network trace, in order to determine the specific packet, and the *tainted* memory block within this packet that is responsible for the attack. In buffer overflow attacks part of the input packet gets copied to a buffer that is too small and the victim program does not perform appropriate bounds checking. If the attack is properly crafted, the input overwrites a pointer value that is past the end of the buffer (Figure 1). As mentioned earlier, *Argos* detects such an attack when this corrupted pointer is about to be used, since that violates security policies. In the correlation phase, we attempt to link M_t to N_t.

Tracking data origins. Before the information correlation step takes place, the collected network traces are preprocessed by reassembling TCP streams. Without reassembly, we would not be able to detect attacks that are split over multiple packets either intentionally, or as a part of TCP fragmentation. Next, rather than scanning the network trace for occurrences of a target value (as done by Covers), we track accurately the origins of tainted memory values, by remembering where they came from in the network trace.

We modified the original *Argos* release to make it track exactly to what byte in the network trace each tainted byte in memory corresponds from the moment it enters the system via the network card. This is not trivial and fairly expensive on the memory as tainted words may be combined. For instance, if two tainted values are added, the destination register should be tainted also and the administration should keep track of the constituent values in the network trace (e.g., as offsets). Obviously, this is only needed for bytes that are tainted and only for the physical memory space given to the *Argos* system. Accurate tracking has been implemented in our prototype as *origin pointers* for each

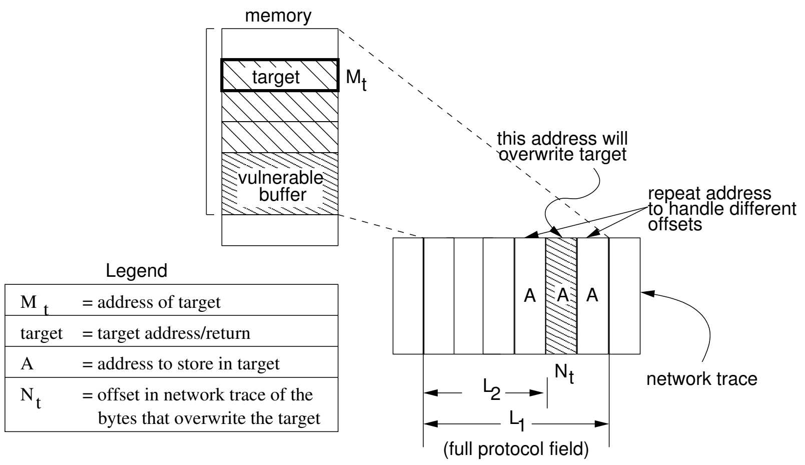
memory
target
M_t
vulnerable buffer
this address will overwrite target
repeat address to handle different offsets
A
A
A
N_t
network trace
L_2
L_1
(full protocol field)
Legend
M_t = address of target
target = target address/return
A = address to store in target
N_t = offset in network trace of the bytes that overwrite the target

Figure 1: Tracing buffer-overflow attacks to protocol fields.

tainted memory location. As instructions take a maximum of 3 operands, each location has a 3-tuple of pointers that each refer to a memory location used to generated the value. More precisely, they refer to the set of origin pointers for that location, which point to other pointers, and so on. The leaves of the tree that is thus formed point to bytes in the network trace.

Multiple fields in a buffer. Sophisticated overflows may be caused by more than one protocol field. An example is chunking and multiple host headers in HTTP, where multiple chunks or headers end up in the same buffer of the vulnerable application. For instance, the Apache-Knacker exploit [13] consists of a GET request containing multiple host headers. The server concatenates the two host fields into one buffer, leading to an overflow. In this case, we need to consider all responsible protocol fields. To the best of our knowledge, we are the first to address this problem. It should be mentioned that the current mechanisms required a substantial modification of *Argos*.

Without going into details, we sketch the mechanism used by *Prospector*. As *Argos* provides not just M_t, the target of the attack, but also all tainted space that was used in the overflow (the shaded area beneath M_t in the figure), we can walk up the stack to see where in the network trace values originate. As a result, we are able to trace all constituent fields that contributed to the overflow. Care was taken to make the method work even if multiple buffers are adjacent on the stack. A potential problem might be caused by the occurence of stale tainted data in the current stack frame, i.e., data that was used in a previous function call and is never untainted. For this reason we make *Argos* clean tainted memory that is stale by untainting a stack frame's residual memory on a return statement.

The mechanism works well for stack-smashing attacks. Heap overflows may be more complex and in rare cases will not be handled properly using the above method due to stale data in allocated chunks (memory chunks that were first allocated and later freed). To remedy this we use a solution for heap-based attacks that is less elegant than for stack-smashing and consists of interposing on the OS's `malloc` functions, to make sure all memory is initialised to zero (automatically untainting stale data). The trick solves the problem of stale tainted data and ensures we can use our mechanism for both heap overflows and double frees, at the cost of a wafer-thin OS-specific wrapper around the memory allocation functions. Summarizing, we pinpoint all packet fragments which cause an alert when copied to a vulnerable buffer in an application.

3.3 Signature Generation

After the preceding step has indicated malicious data in memory and generated a one-to-one mapping with bytes in the network trace, we generate signatures capable of identifying polymorphic buffer overflow attacks. Using knowledge about the protocol governing the malicious traffic, we determine which protocol field contained the data that led to the violation, considering the fields preceding or containing the bytes that caused overflow only. We call these fields *critical.*

Note that vulnerable code usually handles specific protocol fields. Thus, attackers wishing to exploit a certain vulnerability within this code, embed the attack in these protocol fields (or sets of protocol fields in the case of exploits like Apache-Knacker [13]). If values in such fields contain more bytes than can be accommodated by the buffer, an overflow is sure to occur.

3.3.1 Vulnerabilities rather than attacks

We decided to generate signatures by specifying the overflow vulnerability rather than the attack itself. We do so by indicating the protocol fields that should collectively satisfy a condition. In particular, in the current version they should collectively have a length that should not exceed some maximum L_2 (refer to Figure 1) lest they overflow important values in memory (e.g., a function pointer on the heap, or the return address of a function). In the simple case with only one protocol field responsible for the attack, L_2 describes the distance between the beginning of the protocol field and the offset of N_t, containing the value that overwrites the target. Otherwise, L_2 is augmented with the lengths of the remaining critical fields. In both cases L_2 is greater or equal to the length of the vulnerable buffer.

Observe that whenever an application with a given vulnerability receives network data containing the corresponding critical fields with a collective length exceeding L_2 bytes, it will not fit in the application buffer, even if it does not contain any malicious data. Consequently passing it to the application would be inappropriate. In other words, regardless of content, the signatures are unlikely to incur false positives.

Indeed, by focusing on properties like field length, the signatures are independent of the actual content of the exploit and hence resilient to polymorphism. As a result, they can detect different attacks exploiting the same vulnerability, but containing different payloads. Such behavior is quite common, especially if part of the payload is stored in the same vulnerable buffer.

As the signatures generated by *Prospector* identify vulnerabilities, they are application specific. Indeed, we may generate a signature that causes control flow diversion in a specific version of an application, but there is no guarantee that this is also the case for a different version of the same application. In other words, we need precise information about the software to protect. The implication is that the *Prospector* runs at the edge of the network.

The critical fields and the condition that should be satisfied constitute the first, unpolished signature. In practice, however, we want to characterise more precisely what messages constitute an attack. For instance, when the URL field is the critical field that overflows a buffer in a Webserver, it may be that the overflow only works on `GET` requests and not for `POST` requests. In our protocol-specific approach we therefore add a protcol module that determines per protocol which fields may be considered important (e.g., the request type in HTTP) and should therefore be added to the signature. We call such fields *value* fields as explained presently.

3.3.2 The signatures

Every signature consists of a sequence of value fields and critical fields. A value field specifies that a field in the protocol should have this specific value. For instance, in the HTTP protocol a value field may specify that the method should be `GET` for this signature to match, or it could provide the name of a vulnerable Windows `.dll`. Critical fields, on the other hand, should collectively satisfy some condition. For instance, in the current implementation the critical fields should collectively have a length that is less than L_2. Figure 2 illustrates two trivial example signatures.

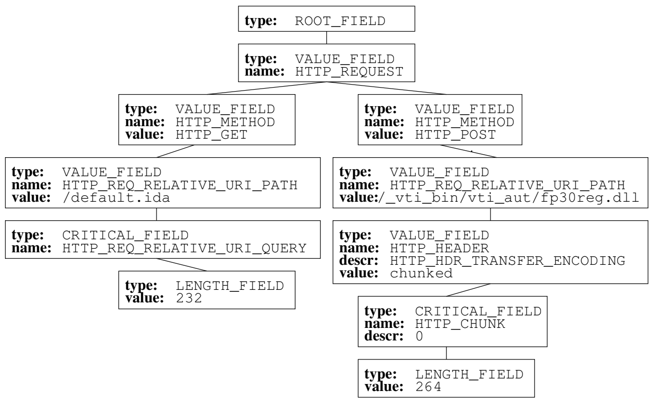
type: ROOT_FIELD
type: VALUE_FIELD name: HTTP_REQUEST
type: VALUE_FIELD name: HTTP_METHOD value: HTTP_GET
type: VALUE_FIELD name: HTTP_METHOD value: HTTP_POST
type: VALUE_FIELD name: HTTP_REQ_RELATIVE_URI_PATH value: /default.ida
type: VALUE_FIELD name: HTTP_REQ_RELATIVE_URI_PATH value:/_vti_bin/vti_aut/fp30reg.dll
type: CRITICAL_FIELD name: HTTP_REQ_RELATIVE_URI_QUERY
type: VALUE_FIELD name: HTTP_HEADER descr: HTTP_HDR_TRANSFER_ENCODING value: chunked
type: LENGTH_FIELD value: 232
type: CRITICAL_FIELD name: HTTP_CHUNK descr: 0
type: LENGTH_FIELD value: 264

Figure 2: Filter tree structure with two signatures

Prospector generates signatures that identify a large class of polymorphic buffer overflows at application-level. Both stack and heap overflows are already handled in the current version. However, given an accurate location of M_t, one may describe format string attacks in a similar way. We do not handle such attacks reliably now, because in the current version address spaces in *Argos* and address spaces of code running without *Argos* are not always equivalent (i.e., there is an offset). The offset stops exploits from taking over the machine and causes no problems for generating signatures for buffer overflow attacks, as the overflow occurs relative to the base of the buffer (and the size of the buffer is obviously the same both with and without *Argos*). Format string exploits, however, may end up overwriting different data than intended, due to their absolute addresses. As a result, *Argos* will detect code injection by format string attacks, but the signatures generated by *Prospector* are currently unreliable. Fixing *Prospector* for format string attacks is future work.

3.4 Intrusion Prevention

We now present our protocol-specific attack prevention method. The *Prospector* IPS contains filters to recognize malicious traffic by matching signatures generated by our system. Unlike the signature generator which may perhaps take tens of seconds to generate a reliable signature, the IPS needs to operate at very high speeds, scanning packets at line rate.

In this section we briefly describe the filter architecture. First, we present its pre-configuration stage. We explain the way multiple signatures are handled, and stored in tree-like structure to check efficiently whether a packet is malicious or legitimate. Next, we show the tree traversing process itself.

3.4.1 Preconfiguration Step

Recall that every signature consists of a sequence of value fields and critical fields. In addition, a signature specifies precisely the vulnerable application, and a condition which the critical fields must collectively satisfy in order to let a packet/flow be classified as malicious. For instance, in the current implementation the critical fields should jointly have a length that is less than L_2 (see Figure 1). While we focus on maximum length in this paper, other conditions could also be imagined (e.g., encoding anomalies, byte distribution).

We organize all the signatures in memory like a tree, so that common prefixes are checked once only. Figure 2 depicts a trivial example consisting of two vulnerabilities. Each significant protocol field is represented by a structure containing the following information: (1) type (e.g., `VALUE_FIELD` or `CRITICAL_FIELD`), (2) field name (e.g., `HTTP_METHOD` or `HTTP_BODY`), and (3) value, relevant only in the case of value fields, can be either a constant, (e.g., `HTTP_GET` or `HTTP_POST`) or a string of characters (e.g., `chunked`).

For efficiency reasons all children of a given node are arranged in increasing order with respect to their name (and value, if needed). This saves time in determining whether a field value in a packet matches a signature.

3.4.2 Processing Packets

We now describe how we check whether a packet matches any of the signatures contained in the tree described in Section 3.4.1. The straightforward solution would be to (1) construct the complete layer 7 message from consecutive TCP segments, (2) dissect this message, (3) traverse the tree to read off whether the message corresponds to any of its branches. However, this approach is rejected for reasons of performance and security. For performance, we cannot afford to copy so much data. For security, we want to discover immediately that the protocol field is too long, rather than holding on until all TCP segments have arrived (by which time the exploit may already have occurred).

Having these restrictions in mind, we built a stateful signature recognition engine. Whenever it obtains a TCP segment, its state is retrieved and dissection is handled by an appropriate protocol dissector. The remaining part of the process is straightforward: with each successive protocol field we descend further down the signature tree using standard BFS to find out whether the message we are handling is malicious.

Note that *Prospector* automatically scans for malformed protocol messages. Since we have protocol-specific knowledge, it was easy to extend *Prospector* to also check whether the application-level interaction conforms to the protocol. In other words, we scrub higher-level protocols.

Since most network traffic is legitimate, it would be desirable to be able to stop dealing with a TCP segment as soon as we realize that it contains benign data. Unfortunately, the length of the whole message is not always known until we completely dissect the stream. Indeed, multiple requests or replies could be sent over a single connection which means we may have to chech the entire stream. For performance reasons we have introduced a configurable optimization into our system. If the shortcut is turned on, and we do not have enough knowledge to determine the length of the remaining part of the layer 7 message, we skip the rest of the *current* TCP segment, provided that we have decided that the current request or reply did not match any signature. Whenever a continuation of this TCP connection comes, we check whether it contains a new request or reply. If not, the segment is also skipped. Otherwise it is passed to an appropriate dissector. Obviously, the optimisation can only be safely applied for servers that also implement the shortcut that a message always starts at a segment boundary.

3.5 Implementation

We implemented the *Prospector*'s filter engine both on a general purpose CPU and on an Intel IXP2400 network processor embedded on a network card. This way we were able to explore different application domains. The CPU implementation is intended to be used as a filter for end-hosts. It protects individual machines and, indeed, individual services, and constitutes a rather extreme form of distributed firewall. The implementation on the IXP2400 was part of an effort to push the distributed firewall a little further into the network, making it harder for end-users to tamper with the system. It was implemented as a line-card solution for an edge router/switch for which application-awareness may be exploited by filtering traffic for a small group of hosts. In addition to *Prospector*, the IXP2400 in this subproject hosts different IDS techniques (e.g., flow-based techniques and regular expression matching).

4 Evaluation

We evaluate *Prospector* along two dimensions: performance and effectiveness. Our experiments were carried out on Intel Pentium M 1.60 GHz CPU with 504 MB main memory. The *Prospector* IPS is implemented additionally on an IXP2400 network processor embedded on a network card.

4.0.1 Signature generation.

The signature generation engine is build on the top of *Ethereal*[2]. During the correlation, it searches through the network trace and reconstructs the byte streams of TCP flows. Note that the logs that are considered by the signature generator are generally fairly short, because we are able to store separate flows in separate files by using our FFPF framework [2]. As a result, *Prospector* may ignore flows that finished a long time ago and flows to ports other than the one(s) reported by forensics. Signature generation times including TCP reassembly for logs of various sizes is shown in Figure 3.

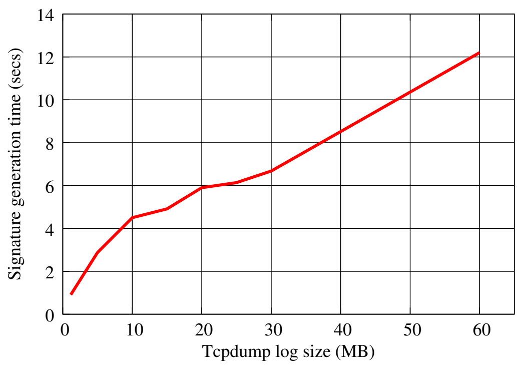

Figure 3: Signature generation times versus log size.

4.0.2 Filtering.

We have benchmarked the *Prospector* IPS both on a general purpose CPU and on the IXP2400's XScale. Figure 4 shows the throughput of *Prospector* and for reference also shows the throughput of a payload-scanning function (we used Snort's Aho-Corasick). We show two versions of *Prospector*: the basic algorithm that touches all data, and a configurable optimized version, called `Prospector+`[3].

Each method processes 4 requests. These are from left to right: a benign `HTTP GET` request that is classified quite easily, a malicious `GET` request that requires much more scanning, and two `POST` requests with bodies of differing lengths. In the malicious `GET` case a significant part of all bytes must be touched (it contains an excessively long `URI`).

However, all three other examples show that if you do not have to touch all bytes – the common case – protocol-deconstruction is more efficient than scanning. Looking at the right-most figures, the longest `POST` request, we see that the gap with aho-Corasick grows quickly as the payload grows. The benign `GET` learns us additionally that skipping remaining headers when a classification has been made can result in a dramatic (here 2-fold) increase in worst-case performance. Note that this example request does not carry a message body. of course, this would be skipped by *Prospector*. Even without message bodies, performance is continuously above 18.000 requests per second on the XScale and above 120.000 requests per second on the Pentium, making the function viable for in-line protection of many common services.

4.1 False positives.

To make sure that *Prospector* does not incur false positives, we manually compared signatures generated by our system with exploits and attack descriptions provided by the Metasploit framework[4]. We also used our signatures to scan a benevolent network trace. Besides homegrown traces, we used the RootFu DEFCON[5] traces that are publicly available for research purposes. We first verified that the exploits we were considering were not present in the traces, by scanning the trace with open source community snort rules, using rules obtained from bleeding snort[6]. Next, we scanned it with the signatures generated by *Prospector*. There were no (false) alerts.

5 Conclusions

We have presented *Prospector*, a protocol specific detector for (possibly polymorphic) buffer overflow attacks. It uses *Argos*, an advanced honeypot to generate signatures for zero-day attacks. In addition, *Prospector* includes a filter engine to block malicious traffic with very few false positives. In the future we plan to address the problem of replaying, for two reasons (1) even more detailed analysis of the attack in a more heavily instrumented *Argos*++ to also catch format string attacks, and (2) to determine all versions of an application vulnerable to a detected attack.

[2]A network protocol analyzer http://www.ethereal.com/.

[3]The optimized version relies on HTTP requests being TCP segment-aligned. Refer to Section 3.4.2

[4]The Metasploit Project http://www.metasploit.com/.

[5]http://www.shmoo.com/cctf/

[6]http://www.bleedingsnort.com

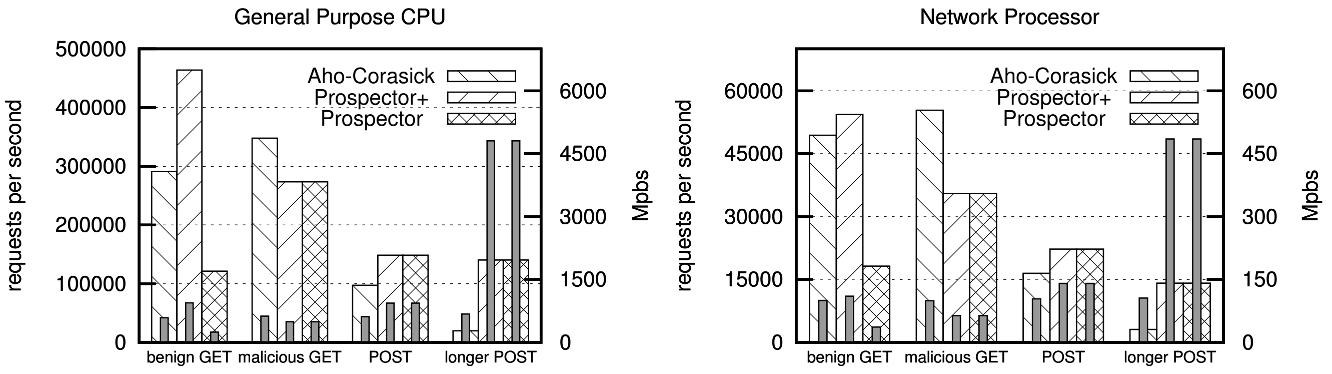

Figure 4: *Prospector* throughput. Wider boxes denote the amount of requests per second (and are scaled by the left Y-axis), while thinner boxes denote the throughput in Mbps (and are scaled by the right Y-axis).

Acknowledgements

This research is sponsored by the EU FP6 'NoAH' project and the Dutch STW 'DeWorm' project. We are grateful to Willem de Bruijn and Tomas hruby for their help with the implementation of the *Prospector* on the network processor.

References

[1] F. Bellard. QEMU, a fast and portable dynamic translator. In *In Proc. of the USENIX Annual Technical Conference*, pages 41–46, April 2005.

[2] H. Bos, W. de Bruijn, M. Cristea, T. Nguyen, and G. Portokalidis. FFPF: Fairly Fast Packet Filters. In *Proc. of OSDI'04*, San Francisco, CA, December 2004.

[3] 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*, Oakland, CA, May 2006.

[4] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford and N. Weaver. Inside the slammer worm. *IEEE Security and Privacy*, 1(4):33–39, July 2003.

[5] B. K. James Newsome and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proc. of the IEEE Symposium on Security and Privacy*, 2005.

[6] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Proc. of RAID'05*, Seattle, USA, Sept 2005.

[7] 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*, Oakland, CA, May 2006.

[8] Z. Liang and R. Sekar. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *Proc. ACM CCS*, pages 213–223, Alexandria, VA, USA, November 2005.

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

[10] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. of the 12th Annual Network and Distributed System Security Symposium (NDSS)*, 2005.

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

[12] W. Robertson, G. Vigna, C. Kruegel, and R. Kemmerer. Using generalization and characterization techniques in the anomaly-based detection of web attacks. In *NDSS'05*, February 2005.

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

[14] V. P. Stuart Staniford and N. Weaver. How to 0wn the internet in your spare time. In *Proc. of the 11th USENIX Security Symposium*, 2002.

[15] M. M. Williamson. Throttling Viruses: Restricting Propagation to Defeat Malicious Mobile Code. In *Proc. of ACSAC Security Conference*, Las Vegas, NV, 2002.