

# Prospector: accurate analysis of heap and stack overflows by means of age stamps

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

## Abstract

Heap and stack buffer overflows are still among the most common attack vectors in intrusion attempts. In this paper, we ask a simple question that is surprisingly difficult to answer: which bytes contributed to the overflow? By careful observation of all scenarios that may occur in overflows, we identified the information that needs to be tracked to pinpoint the offending bytes. There are many reasons why this is a hard problem. For instance, by the time an overflow is detected some of the bytes may already have been overwritten, creating gaps. Additionally, it is hard to tell the offending bytes apart from unrelated network data. In our solution, we tag data from the network with an age stamp whenever it is written to a buffer. Doing so allows us to distinguish between different bytes and ignore gaps, and provide precise analysis of the offending bytes. By tracing these bytes to protocol fields, we obtain accurate signatures that cater to polymorphic attacks.

## 1 Introduction

Polymorphic network attacks are difficult to detect and even harder to fingerprint and stop. This is especially true if the exploit itself is polymorphic [12]. Fingerprinting is the process of finding out how an attack works, i.e., what an attacker should do to make the exploit succeed. It is important for two reasons: analysis of the attack (e.g., by human security experts), and signature generation.

Signature generation is hard because of the complex and conflicting list of constraints. First, signatures should incur a negligible ratio of false positives. Second, the number of false negatives should be low. Third, we should be able to check signatures at high rates. Fourth, we should cater to polymorphic attacks with polymorphic exploits. In our work we further aim for fast, one-shot generation without the need to replay the attack.

In this paper, we address the problem of polymorphic *buffer overflow* attacks on *heap* and *stack*. Given their long history and the wealth of counter-measures, it is perhaps surprising that buffer overflows are still the most popular attack vector. For instance, more than one third of *all* vulnerabilities notes reported by US-CERT in 2006 consisted of buffer overflows [33]. As the US-CERT's database contains many different types of vulnerabilities (leading to denial of service, privacy violation, malfunctioning, etc.), the percentage of buffer overflows in the set of vulnerabilities leading to *control* over the victim is likely to be higher. Even Windows Vista, a new OS with overflow protection built into the core of the system, has shown to be vulnerable to such attacks [27].

Polymorphic attacks demand that signature generators take into account properties other than simple byte patterns. For instance, previous approaches have examined such properties as the structure of executable data [16], or anomalies in process/network behavior [10, 18, 21].

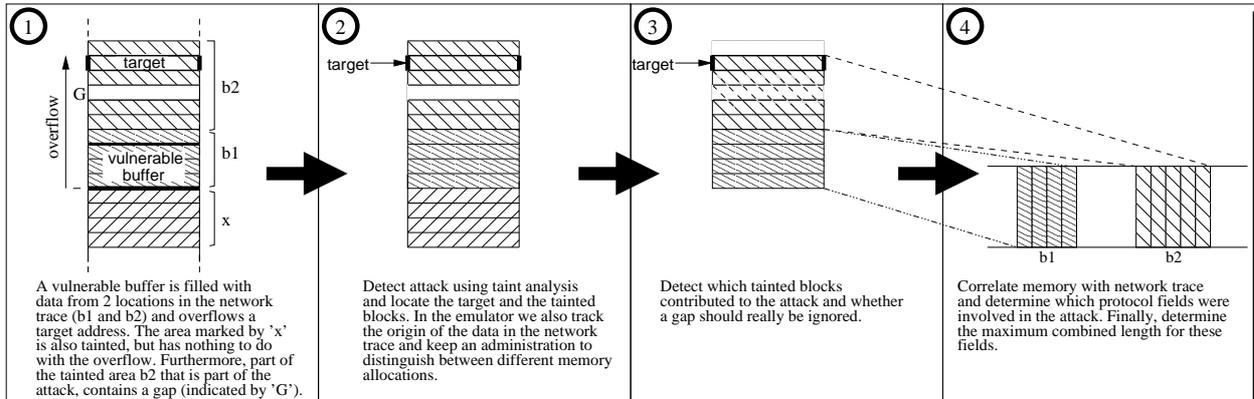


Figure 1: Main steps in *Prospector's* attack analysis.

In contrast, in this work we asked a simple question that is surprisingly hard to answer: what bytes contribute to an attack? As we will see, an answer to this question also trivially yields reliable signatures that meet the requirements listed earlier. Like [4], we focus on the vulnerabilities rather than specific attacks, which makes the signatures impervious to polymorphism. The full system is known as *Prospector*, a protocol-specific detector of polymorphic buffer overflows. It deals with both heap and stack overflows in either the kernel or user processes and while it was implemented and evaluated on Linux, the techniques apply to other OSs also.

In a nutshell, the idea is as follows (see also Figure 1). We use an emulator-based honeypot with dynamic taint analysis [26] to detect attacks and to locate both the exact address where a control flow diversion occurs and all the memory blocks that originate in the network (known as the *tainted* bytes). The emulator marks all bytes originating in the network as tainted, and whenever the bytes are copied to memory or registers, the new location is tainted also. We trigger an alert whenever the use of such data violates security policies (e.g., jumping to tainted data).

Next, we track which of the tainted bytes took part in the attack. For instance, in a stack overflow we walk up the stack looking for tainted bytes. However, we must weed out all the memory that, while tainted, had nothing to do with the attack (e.g., *stale* data that was part of an old stack frame, such as the bytes marked *x* in the figure). To do so, we track the *age* of data at runtime, so that we know whether memory on the heap or stack is a left-over from an older allocation. As a result, we can distinguish between relevant bytes and memory to be ignored.

Once we know which bytes were in the buffer overflow and we can trace them to the bytes that arrived from the network, we can easily find out which protocol fields contributed to the attack. If  $n$  fields were involved in the overflow with a *combined* length of  $N$ , we know that any similar protocol message with a combined length for these fields greater or equal to  $N$  will also lead to a buffer overflow. Using the maximum length of a (single) protocol field as the signature of a polymorphic attack was first proposed in Covers [20]. While Covers may be considered an inspiration for our work, we will see that it suffers from both false positives and false negatives. In this sense our signature generator remedies and extends the Covers technique.

Still, this method is inappropriate for attacks based on messages that contain a specially crafted (wrong) length field, misspecifying the length of another protocol field. As we will see, to detect such attacks, the signature pinpoints the length field and specifies when misbehavior occurs.

We emphasize, however, that the main contribution of this work is the pinpointing of which bytes contribute to an overflow. The signature generator is intended to demonstrate the usefulness of such data in practice. While the end result is a fairly powerful signature generator in its own

right, very different signature generators could also be built on this technique. In addition, it could generate a wealth of information for human security experts.

In summary, our contributions are:

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

There are other contributions as well (e.g., a novel way to monitor process switches from an underlying emulator), but as they are not the focus of this paper, we will not dwell on them in this section, and defer the discussion to the relevant sections. Finally, we extended *Prospector* with an attack vector-specific module to make it deal with double free attacks.

Besides the functionality, of course, one of the main questions concerns performance: is the tracking fast enough to be of practical use? While emulation, taint analysis, and age tracking all incur a fair amount of overhead, we believe that *Prospector* is well-suited for honeypots. Indeed, the slow-down compared to the *Argos* honeypot on which it is based is less than 20%. For full taint-analysis, *Argos* is considered a fast emulator, so we believe the overhead is acceptable.

The remainder of this paper is organized as follows. In Section 2, we place our work in the context of related work. Section 3 discusses heap and stack overflows and highlights factors that complicate the analysis. Sections 4 and 5 describe the design and implementation of *Prospector*, respectively. The system is evaluated in Section 6, and conclusions are drawn in Section 7.

## 2 Related work

Recent worms have started using polymorphic engines like ADMmutate [19]. They work by inserting garbage and NOP insertions, and/or by substituting code by equivalent code, shuffling registers, and encryption. While more constrained, even exploits are made polymorphic [12].

Previous work on detection of polymorphic attacks included techniques that looked for executable code in the messages. Techniques include: (a) abstract or actual execution of network data in an attempt to determine how long the maximum executable length of the payload would be [32], (b) static analysis to detect exploit code [5], (c) sled detection [1], and (d) structural analysis of binary to find similarities between worm instances [16].

Taint-analysis has been used in several projects for signature generation [23, 7]. However, none of the existing projects provide an answer to the question of which bytes were involved. Enhanced tainting [3] expands the scope of tainting to also detect such attacks as SQL injection and XSS, but requires source code transformation.

Transport-layer filters independent of exploit code are proposed in Shield [34] with signatures in the form of partial state machines modeling the vulnerability. Specific protection against instruction and register shuffling, as well as against garbage insertion is offered by semantics-aware detection [6].

A related project, PolyGraph [22], fingerprints attacks by looking at invariant substrings present in different instances of suspicious traffic. The idea is to use these substrings as a signature. Such methods are vulnerable to the injection of noise in the data stream [25].

Various groups have proposed anomaly detection for catching polymorphic attacks. PAYL [35] builds a model of the byte distribution of normal traffic and compares real traffic with this model. Increasingly sophisticated mimicry attacks [15, 13] are a problem and spark many new developments in this direction [11, 10, 17].

SigFree [37] observes that overflow attacks typically contain executables whereas legitimate client requests never contain executables, and blocks attacks by detecting the presence of code.

```

[1] void read_from_socket (int fd) { // fd is the socket descriptor
[2]     int n;
[3]     char vuln_buf [8];          // the vulnerable buffer
[4]     char unrelated [8];        // a safe buffer, unrelated to the attack
[5]     read (vuln_buf, fd, 32);    // from socket: taints all data in 'vuln_buf' and above (overflow possible)
[6]     read (unrelated, fd, 8);    // from socket: taints all data in 'unrelated' (no overflow possible)
[7]     n = 1;                      // untaints 4 bytes of data that was previously tainted, creating a gap
[8]     return;
[9] }

```

Figure 2: Tainted data: gaps and dirt (unrelated tainted data).

Brumley et al. propose vulnerability-based signatures [4] that match a set of inputs (strings) satisfying a vulnerability condition (a specification of a particular type of program bug) in the program. When furnished with the program, the exploit string, a vulnerability condition, and the execution trace, the analysis creates the vulnerability signature for different representations: Turing machine signatures, symbolic constraint signatures, and regular expression signatures.

Packet Vaccine [36] detects anomalous payloads, e.g., a byte sequence resembling a jump address, and randomizes it. Thus, exploits trigger an exception in a vulnerable program. Next, it determines information about attack (e.g., the corrupted pointer and its location in the packet) and generates a signature much like Covers [20]. Signatures are subsequently refined by trying variations of the vaccine. Packet Vaccine suffers from the same problems as Covers. It neither checks for multiple separate fields, nor worries about the granularity of the protocol dissector, leading to false positives and negatives.

Dynamic protocol analysis [9] proposes the design of dynamic application-layer protocol dissection to deal with remote attacks that try to not use a standard port to evade security measures based on protocol-analyzers. Our current signature generation engine is based on Ethereal. However, if necessary, we could easily port it to any other network protocol analyzer.

### 3 Attacks and factors complicating the analysis

*Prospector* caters to both heap and stack overflows. *Stack* overflows are conceptually simple. Even so, they prove to be hard to analyze automatically. Essentially, a vulnerable buffer on the stack is overflowed with network data until it overwrites a target that may lead to control flow diversion (typically the return address). An important observation here is that the data that is used for the overflow may originate in more than one set of bytes in the network flow. In Figure 1 this is illustrated by regions *b1* and *b2*, respectively.

Taking into account either fewer or more protocol fields may lead both to false positives and negatives. Covers [20], by using a single protocol field, therefore lacks accuracy in a multi-field attack. In addition, it suffers from possible granularity differences between application and protocol analyzer (Section 4.8.1). We show later that handling multi-field attacks is much more complicated.

The naive solution for finding the bytes that contribute to the attack is to start at the point of attack (the *target* in Figure 1) and grab every tainted byte below that address until we hit an area that is not tainted. Unfortunately, while all bytes that contributed to the attack were tainted at some point, there are complicating factors that prevent us from using such a naive solution. First, there may be *gaps* in the tainted block of memory that was used in the attack. For instance, the code in Figure 2 may lead to a gap, because the assignment to *n* occurs after the overflow.

Second, the naive solution gathers tainted blocks that are unrelated to the attack. An example is the region marked by *x* in Figure 1. It may be caused by left-over data tainted from an old stack frame, or by safe buffers adjacent to the vulnerable buffer, such as the buffer `unrelated` in

Figure 2. In this paper, we will informally refer to such unrelated tainted data as *dirt*.

Heap corruption can be more complex than a stack overflow and potentially more powerful. A *heap* corruption attack can take two main forms. A simple overflow occurs when critical data (e.g., a function pointer) is overwritten from a neighboring chunk of memory, or from another field of a structure. In a more *advanced* form, the attacker overflows link pointers that are used to maintain a structure keeping free regions. It allows an attacker to overwrite virtually any memory location with any data [2]. The problem is caused by the implementation of memory allocation functions which store control data together with the actual allocated memory, thus providing attackers potential access to information used by the operating system memory management.

The problem of gaps and dirt also exists for heaps and is mostly similar to that of the stack. For heap overflows, instead of the occurrence of stale tainted data from a previous function call, we may encounter stale tainted data used in a previous function that allocated the memory region. In addition, we need to deal with possible adjacent fields of a structure.

*Advanced* heap corruption attacks yield an additional complication. Since the attacker can overwrite any memory location with any contents, it is possible that at detection time the memory region which was holding the vulnerable buffer is reused and contains unrelated data. If left unhandled, such a scenario would prevent us from pin-pointing exactly the data responsible for the intrusion attempt.

Finally, numerous protocols have fields specifying the length of another field, say  $l_f$  defining the length of field  $f$ . Attackers may manipulate this length value, and via heap overflow take control of the host. First, a malicious message provides  $l_1$  instead of  $l_f$ , with  $l_1 \gg l_f$  and close to the maximum size of an integer. The application allocates  $l = l_1 + k$  bytes (where  $k$  bytes are needed to store some application-specific data), and  $l$  ends up being a small number because of the integer wrap-around,  $l \ll l_1$ . As a result, the application copies  $l_1$  bytes into the buffer leading to overflow. In the second scenario, rarely seen in the wild, the attacker provides  $l_2$  smaller than expected,  $l_2 < l_f$ , the application allocates a buffer of size  $l_2$  which is not sufficient to hold the data, and a subsequent copy operation without boundary checks spills network data over adjacent memory. Notice that we cannot draw any conclusions about a message containing such attacks by relying only on the observation that  $n$  fields were involved in the overflow with a combined length of  $N$ .

We conclude this section with an assumption. We assume that overflows occur by writing bytes beyond the high end of the buffer. While not strictly necessary, it makes the explanation of our work easier. However, the techniques described in this paper can be trivially extended to handle the reverse direction also (i.e., attacks that overwrite memory below the start of a vulnerable buffer).

## 4 Design

The main steps of *Prospector*'s attack analysis are sketched in Figure 1. In this section, we first describe how we instrument the execution and what data is produced by our taint-analysis emulator (Sections 4.1–4.5). From this data we formally derive a set of properties of tainted regions in Section 4.6. We then show in Section 4.7 how we use these properties to determine the exact bytes in the attack. The memory that constitutes these bytes will be referred to as the *crucial* region. Finally, we correlate the information with protocol fields in network data to obtain signatures.

### 4.1 Argos

*Prospector* uses *Argos* [26] to detect attacks. *Argos* is an efficient and reliable hardware emulator that uses taint analysis [23] to tag and track network data. Data originating in the network is marked as tainted, and whenever it is copied to memory or registers, the new location is tainted

also. *Argos* raises an alert whenever the use of such data violates security policies. To aid signature generation *Argos* dumps the content of all registers, as well as tainted memory blocks to file, with markers specifying the address that triggered the violation, the memory area it was pointing to, etc. In addition, *Argos* allows us to keep track of the exact origin of a tainted memory area, in the form of an offset from the start of the network trace. In practice, the offset is used as (32 bit) tag.

Even with such accurate administration of offsets, the problem of identifying the crucial regions remains. To handle it, we have extended *Argos* in various ways. In the next few sections we will explain the blocks that together form our information correlation engine. We start with support for an advanced heap corruption attack, and then explain how we pinpoint the relevant tainted memory region.

## 4.2 Dealing with advanced heap overflows.

In the case of stack overflows and simple heap corruption attacks, we know from where to look for the crucial regions: in the memory area beneath the violation address reported by *Argos*. In contrast, advanced heap corruption attacks, require us to find first the memory region containing the vulnerable buffer. Only then we can start marking the bytes that contributed to the attack.

As mentioned earlier, such attacks may easily lead to a situation in which at detection time, the memory region that was holding the vulnerable buffer is reused and contains unrelated data. To deal with this scenario, the emulator marks the bytes surrounding an allocated chunk of memory as *red*. If tainted data is written to a red region, indicating an overflow (although not necessarily an attack, see also Section 5.2), we keep the application running, but we dump the memory region covering the whole vulnerable buffer for potential later use, i.e., we traverse down the heap, storing data until we come across a red marker indicating the beginning of the vulnerable buffer. This works as common memory management systems store control data in-line together with allocated chunks of memory. Consequently the bytes surrounding an allocated buffer contain control data, which should never be overwritten with data coming from the network.

In the case of an intrusion attempt, we search for the violation address and the network index in the dumped heap areas in order to find a memory region containing the buffer that contributed to the attack. Having these pieces of memory we perform further analysis to overcome all the difficulties described in Section 3. However, we emphasize that it is highly unlikely that we would suffer from lack of data, since we dump the tainted memory exactly in the moment of overflow. The red markers are vaguely reminiscent of StackGuard's canary values [8], but different in that they are maintained by the emulator and trigger action immediately when they are overwritten.

## 4.3 Dealing with malformed messages in heap overflows.

To cater to heap corruption attacks founded on malformed length field in a message, we check whether allocating a chunk of memory relies on remote data. Whenever an application calls `malloc(size)` with the `size` variable being tainted, we associate the network origins of the length parameter with the new memory chunk. In the case of an intrusion attempt, it enables us to understand the reasons for failure, and generate a correct signature. (Section 4.8.1.)

## 4.4 Age stamps.

In order to distinguish between stale and relevant data both on stack and heap we introduce an *age stamp* indicating the relative age of data regions. `AgeStamp` is a global counter, common to the entire OS running on the *Argos* emulator. The need for a system-wide global variable stems from the fact that memory may be shared.

`AgeStamp` is increased whenever a function is called (a new stack frame is allocated) or returns. To be precise, we update `AgeStamp`  $v_1$  to  $(v_1 + 1)$  only if in period  $v_1$  a tainted value was stored in the memory. Otherwise it is not necessary, as we shall soon see. If a tainted value is copied to memory, we associate the current `AgeStamp` with the destination memory location, i.e., for each tainted value we remember the ‘time period’ in which it was stored. In addition, for each process and lightweight process we allocate a history buffer, where we store information about allocation and release of stack frames, as follows: for each function call and return we store the value pair (stack pointer, `AgeStamp`). When an application allocates a buffer on the heap, we associate the current `AgeStamp` with this memory region. When a memory field becomes untainted, we do not clean the age stamp value.

Observe that the order of age stamps in the crucial region right after the overflow (before gaps appear) is nondecreasing. We will use this observation in the analysis phase in Section 4.7 to spot tainted bytes stored later than the crucial tainted memory region, and so forming a gap, or dirt. For instance, the `unrelated` buffer in Figure 2 has the corresponding age stamps bigger than `vuln_buf`, and so we can conclude that it does not belong to the crucial tainted memory region.

The explanation of the observation is straightforward. If the buffer was overwritten with one call to a copying function, all tainted bytes have the same age stamp, and so the observation holds. Otherwise, the observation results from the assumption that buffers overflow from low to high addresses. Indeed, if the lower part of the buffer was filled by a function `fun1` in the time period `AgeStamp1`, and later on the higher part - by a function `fun2` in the period `AgeStamp2`, then `AgeStamp1 < AgeStamp2`.

#### 4.5 Additional indicators.

Even though age stamps support separation of dirty areas, they are not powerful enough. Coming back to Figure 2, imagine that instead of calls to the `read` function in lines 5 and 6, the application copies data without any calls and thus without the increment of time period, e.g., by two `while` loops. Then we would not have means of classifying the `unrelated` buffer as unnecessary, since `vuln_buf` and `unrelated` would share the value of `AgeStamp`. To remedy this situation we introduce two extra indicators which let us establish the order in which the two given buffers were filled.

Associated with each memory location in the guest OS are two one-bit markers that serve to keep additional information about bytes copied to memory. They are known as `prev_fresh_taint` and `1st_taint_next_address`, respectively. Intuitively, the former indicates for tainted address  $a$  whether  $a - 1$  was tainted and if so, whether this happened more recently. The latter indicates that the tainted store at  $a$  was the first such store to  $a$  after  $a - 1$  was tainted. However, their exact meaning are defined below. We first introduce the indicators. Then, in Section 4.6 we formally list a few properties that hold for the tainted regions, but the real use will be explained in Section 4.7.

- `prev_fresh_taint`. Whenever a tainted value is stored in memory location `addr1`, we perform the following assignment: `prev_fresh_taint(addr1 + 1) = 1`. From `(addr1 + 1)`’s point of view it means that the address below has freshly tainted contents, more recent than its own.
- `1st_taint_next_address`. For each tainted value stored in memory location `addr1`, we perform the following: if `prev_fresh_taint(addr1)` is set then `1st_taint_next_address(addr1) = 1`, and `prev_fresh_taint(addr1) = 0`. It means that the value at `addr1` is the result of the first tainted data store operation after the memory location beneath it became tainted. If memory location `addr1` is updated more than once without change of `(addr1 - 1)`, then `1st_taint_next_address(addr1)` becomes 0.

- Whenever a tainted value is copied to memory location  $\text{addr}_1$ ,  $\text{1st\_taint\_next\_address}(\text{addr}_1)$  becomes 1, and  $\text{AgeStamp}(\text{addr}_1 - 1) < \text{AgeStamp}(\text{addr}_1)$ , we store the age stamp of a memory location beneath ( $\text{addr}_1 - 1$ ) for possible later use. Intuitively, this happens when the memory location  $\text{addr}_1$  is the first byte of a buffer which was copied by a particular function, and no tainted data was stored here since the memory location beneath it became tainted. Since it can easily be a beginning of a new buffer adjacent to an existing tainted region, we cannot afford to miss it. The exact reasons will become clear soon. In this paper, we will refer to this age stamp as *stored age stamp*.
- When a memory location becomes untainted, we do not touch the values of the  $\text{1st\_taint\_next\_address}$  and  $\text{prev\_fresh\_taint}$  markers.

#### 4.6 Formal specification of the properties of tainted data and gaps

In this section, we use the indicators defined above to derive properties of regions of tainted memory.

**Observation 4.1** *Let  $\text{buf}$  be a crucial tainted region of size  $n$ . Then*

1.  $\forall i = 0 \dots (n - 1)$   $\text{buf}[i]$  is tainted,
2.  $\forall i = 0 \dots (n - 1)$   $\text{AgeStamp}_i \geq \text{alloc\_age\_stamp}$ , where  $\text{alloc\_age\_stamp}$  denotes the time period in which the given buffer was allocated,
3.  $\forall i, j = 0 \dots (n - 1), i < j$ ,  $\text{AgeStamp}_i \leq \text{AgeStamp}_j$ ,
4.  $\text{1st\_taint\_next\_address}(\text{buf}[0])$  depends on the taintedness of the address below,
5.  $\forall i = 1 \dots (n - 1)$   $\text{buf}[i]$  finds its  $\text{prev\_fresh\_taint}$  indicator set to 1, so it ends up with  $\text{prev\_fresh\_taint}$  set to 0, and  $\text{1st\_taint\_next\_address}$  to 1.

We now deal with discontinuities such as gaps of non-tainted data in an otherwise tainted region of memory that contributes to the attack.

**Observation 4.2** *Let  $\text{gap}$  be a non-tainted discontinuity located inside a crucial tainted memory region  $\text{buf}$ , i.e., a region in  $\text{buf}$  where property (1) from Observation 4.1 does not hold. Since neither age stamps nor indicators are changed when a memory location becomes untainted, the remaining characteristics of  $\text{buf}$ , (2) - (5) from Observation 4.1, are satisfied within  $\text{gap}$ .*

A second type of discontinuity consists of a gap containing ‘dirt’ (unrelated tainted data) in a tainted memory region that contributes to the attack.

**Observation 4.3** *Let  $\text{gap}$  be a tainted discontinuity of size  $m$  located inside a crucial tainted memory region  $\text{buf}$ . Then*

1.  $\forall i = 0 \dots (m - 1)$   $\text{gap}[i]$  is tainted,
2.  $\forall i = 0 \dots (m - 1)$   $\text{AgeStamp}(\text{gap}[i]) \geq \text{AgeStamp}(\text{location just above gap})$ ,
3. The memory location in  $\text{buf}$  just above  $\text{gap}$  has both indicators  $\text{prev\_fresh\_taint}$  and  $\text{1st\_taint\_next\_address}$  set to 1, while  $\text{gap}[0]$  has both indicators set to 0.
4.  $\forall i = 1 \dots (m - 1)$   $\text{gap}[i]$  finds its  $\text{prev\_fresh\_taint}$  indicator set to 1, so it ends up with  $\text{prev\_fresh\_taint}$  set to 0, and  $\text{1st\_taint\_next\_address}$  to 1.
5. All memory locations within  $\text{buf}$  not listed in (1)-(4) of this remark retain original values of their age stamps and indicators.

Of course, a gap containing dirt may adjoin a similar gap. In that case, they simply merge as follows. If  $\text{gap}$  is a tainted discontinuity located inside a crucial region  $\text{buf}$ , and the bottom (top) part of  $\text{gap}$  adjoins another tainted discontinuity  $\text{gap}_b$  ( $\text{gap}_t$ ), then both holes merge together forming a single discontinuity for which all properties listed in Observation 4.3 hold.

## 4.7 Analysis

To find the bytes that contributed to the attack (the crucial region), we traverse the memory downwards starting at the violation address and continue as long as the bytes we come across conform to Observation 4.1. In this Section we discuss how to start this process and how to overcome the complicating factors mentioned in Section 3.

**The age of allocation.** We start the analysis by figuring out `alloc_age_stamp`, the age (or ‘time period’) in which the vulnerable buffer containing the violation address reported by *Argos* was allocated. We need it to distinguish between data stored before and after the buffer was allocated.

In the case of a heap corruption attack, we have this age stamp simply correlated with each chunk of memory. In a stack smashing attack (i.e., when the violation address is not smaller than the value of the stack pointer register `ESP`), we check the history of stack frames associated with the vulnerable process for the most recent entry above the violation address. If the malicious data was spilled over the adjacent stack frame as well, we may find an age stamp of a caller function instead. However this does not prevent the correct analysis, because when we start looking for the whole crucial region later, we will figure out the most recent, and proper `AgeStamp`.

While incrementing an age stamp for a function call is fairly intuitive (and seems to correspond to keeping track of ‘time’ through function calls), incrementing it for function *returns* is perhaps less clear. We now explain why we need the history of age stamps for returns (`ret` instructions) as well. It should be mentioned that if all programs allocated all local variables at the beginning of a function, we would not have to care about the `ret` instruction. Suppose, however, the following occurs: (1) function `fun1` calls `fun2`, (2) `fun2` stores tainted data on the stack, and associates `AgeStamp2` with it, (3) `fun2` returns, (4) `fun1` allocates local buffer `buf`, which ends up in the stale `fun2` function frame. If we want to correctly distinguish between data stored before and after `buf` was allocated, we cannot rely on the age stamp associated with `fun1`’s function frame. In that case, we would not be able to remove the stale bytes left by `fun2` from our analysis. By also tracking the `ret` instruction, we correctly conclude that `buf` was allocated in an age following `AgeStamp2`.

**Gaps.** One of the difficulties identified in Section 3 concerned gaps in the crucial region’s tainted data. Such discontinuities occur for instance when the program assigns a new value to a local variable allocated in the crucial region after the overflow took place. They can also arise if the whole or parts of the vulnerable buffer are refilled by the application. Let us assume for now that the discontinuity is fully included in the crucial tainted memory region, i.e., it does not reach the bottom. In other words, below the gap there is at least one byte which contributed to the attack.

Again, to find the crucial region we traverse the memory as long as the bytes encountered are tainted in accordance with Observation 4.1. However, we now come across a non-tainted or tainted discontinuity before we reach the region’s bottom. To handle such gaps, we look for the end of the discontinuity to find out how many bytes of the crucial region we are missing.

If we find a byte at variance only with property (1) of Observation 4.1 (i.e., it is not tainted), we conclude that it belongs to a non-tainted discontinuity. In order to determine the end of the gap, its length, and the remains of the vulnerable buffer `buf`, we traverse the memory further until we encounter tainted data (see Observation 4.2). Since we assume that the gap does not reach the beginning of the vulnerable buffer, we will eventually spot a tainted byte. In the worst case, this will be `buf[0]`.

We can also find a byte in memory location `addr1` at variance with property (5) or properties (3) and (5) of Observation 4.1, where inconsistency with (5) means that `prev_fresh_taint(addr1)` is equal to 1. Observation 4.3 says that it is probable that we have just spotted a tainted discontinuity.

There are two scenarios possible: (i) the discontinuity was caused in the same age as the buffer overflow, and property (3) of Remark 4.1 holds; (ii) the gap was established in a later period, so property (3) is not fulfilled. Analogous to the case of non-tainted discontinuity, we now traverse the memory until we encounter a memory location with the two indicators not set, in order to determine the end of the hole, its length, and the remains of the vulnerable buffer `buf` (see Observation 4.3). Again, as in the previous paragraph, we are certain to spot the end of the gap.

Summarizing, we know how to detect boundaries of a discontinuity established in a crucial memory region. Note that without the extra indicators we would not be able to identify the tainted gap established in the same age as the remains of the vulnerable buffer.

**Excess of data.** We now discuss how to determine the beginning of the vulnerable buffer `buf`, thus we address the problem of dirt. For the sake of simplicity, we again assume that there is no discontinuity at the beginning of `buf`. It means that at the point of intrusion detection, `buf[0]` contains the byte that contributed to the attack. By traversing the memory we eventually encounter this byte.

Consider the successive possible instances of the beginning of the vulnerable buffer. For each of the scenarios we explicitly discuss the contents of essential variables at the time of the overflow and at the time of detecting the intrusion. For the sake of clarity let us denote the memory location of `buf[0]` by `addrB`, and the address below `buf[0]` by `addrA`. We assume that traversing the memory as discussed above led us to byte `addrB`, and we check whether appropriate conclusions enabling to spot correctly the buffer boundary can be drawn.

1. **Overflow:** `prev_fresh_taint(addrB)` is equal to 0 (then `addrA` is untainted); we set `1st_taint_next_address(addrB) = 0`.  
**Detection:** we encounter a byte with `1st_taint_next_address` set to 0, and we are not inside a discontinuity. We conclude that at the time of the overflow `addrA` was untainted. Thus we have just encountered the beginning of `buf`. To make the conclusion clear, note that inside a tainted vulnerable buffer there is only one possibility for a byte to have the `1st_taint_next_address` indicator set to 0, namely at the beginning of a gap (Observations 4.1 and 4.3.)
2. **Overflow:** `prev_fresh_taint(addrB)` is equal to 1, but `addrA` contains stale tainted data; we set `1st_taint_next_address(addrB) = 1`.  
**Detection:** We encounter a byte with `1st_taint_next_address` set to 1, which has the stored age stamp of memory location beneath associated. We compare this age stamp with `alloc_age_stamp` of the vulnerable buffer `buf` to conclude that at the time of overflow `addrA`'s value was stale, so we have just encountered the beginning of `buf`.
3. **Overflow:** `prev_fresh_taint(addrB)` is equal to 1, `addrA` contains fresh tainted data; we set `1st_taint_next_address(addrB) = 1`. Since `addrA` merged with `buf` form together an area that conforms to all the properties of a crucial region (see Observation 4.1), we will treat `addrA` as a part of the tainted buffer we are looking for. Note that we cannot detect that `addrA` belongs to a distinct variable. Most compilers (including `gcc`) allocate stack memory for a few local variables at once, making it impossible to see the boundaries between successive buffers. Similarly, on the heap, memory is allocated for a structure as a whole, rather than for the individual fields separately.  
**Detection:** We come across a byte with `1st_taint_next_address` set to 1. Regardless of the existence of the stored age stamp of the memory location below it, we will conclude that at the moment of overflow `addrA`'s value was fresh, and so is supposed to belong to the vulnerable

buffer. We will simply go on with the analysis looking for gaps and end of the buffer as if nothing had happened. Depending on the application behavior between the moment of overflow and that of detection, we will end up either adding dirty data to the crucial tainted memory region or spotting a contradiction with Observations 4.1-4.3 and reversing to the last correct byte encountered,  $\text{addr}_B$ . The first possibility comes true, if we reach a buffer that is totally filled with network data, the possible area between this buffer and  $\text{addr}_B$  pretends to be a dirty gap, and additionally the whole region containing the dirty buffer, the dirty gap, and the crucial tainted memory region is in accordance with Observations 4.1-4.3. Note however, that even in this unlikely case we could only incur false negatives, and never false positives, since the dirty buffer needs to be filled totally.

We have not discussed what happens if the discontinuity in the vulnerable buffer reaches the buffer's bottom. In principle, the analysis is analogous to the one presented above. What is worth noting, is the fact that we cannot determine the length of the discontinuity and we may miss part of the crucial tainted memory region, since the bottom part of the vulnerable buffer gets overwritten.

## 4.8 Signature Generation

After the preceding steps have identified the 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 first list the protocol fields including the crucial tainted memory region. Due to possible excess of tainted data in rare scenarios described in Section 4.7, we include a protocol field in a signature either if it contains the violation address, or if a cohesive part of it including at least one boundary can be mapped to the indicated malicious data. 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 [30]). If values in such fields contain more bytes than can be accommodated by the buffer, an overflow is sure to occur.

### 4.8.1 Vulnerabilities rather than attacks

We generate signatures for stack and heap overflows by specifying the 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 the signature specifies that the fields should collectively have a length  $L$  that does not exceed some maximum, 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,  $L$  describes the distance between the beginning of the protocol field and the position in the network trace that contains the value that overwrites the target. Otherwise,  $L$  is augmented with the lengths of the remaining critical fields. In both cases  $L$  is greater or equal to the length of the vulnerable buffer. Signatures can be checked by a protocol dissector (similar to Ethereal) that yields the fields in a flow.

**Heap overflows founded on malformed length.** As mentioned earlier, signatures for heap corruption attempts that manipulate a length field need to relate the critical fields to the length field. Thus, after having determined the crucial tainted memory region  $\text{buf}$  of length  $l$ , we check in the network trace for the length value provided by the attacker  $l_a$ . If it is bigger than  $l$ , we specify that a message contains an attack if the cumulative length of the critical fields is smaller than  $l_a$

with the length field greater or equal  $l_a$ . In the second scenario, with  $l_a < l$ , we must be more cautious, since the value provided by the attacker does not need to define the number of bytes, but it could describe amount of integers or any other structures. For now we describe the malicious message similarly as in the case of overflows regarding static-length buffers, requiring conformity of the length value with the actual size of the protocol fields. Thus as a value for  $L$  we provide the length field. To assure that the signature is indeed correct we need to verify it by checking whether *Prospector* or *Argos* spot an illegal operation if we send a message with critical fields filled with arbitrary bytes in the size slightly exceeding `length_field`. If it appears that we are wrong, the only thing we can do is use the semantics of the protocol for a description of the length field.

**Multiple fields.** By handling multiple fields, *Prospector* fixes and generalizes the signature generation in Covers [20]. Also, unlike [20], we do not require the protocol dissector to match the granularity in which the application works with protocol messages. The granularity of the dissector may be larger or smaller than that of the application. For instance, the dissector may indicate that a message contains two fields  $F1$  and  $F2$ , while the application copies them in one in a single buffer in one go (essentially treating them as a single field  $F$ ).

**False positives.** Observe that whenever an application with a given vulnerability receives network data containing the corresponding critical fields with a collective length exceeding  $L$  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 will not incur false positives in practice. However, in an unlikely scenario it is possible that we cannot correctly determine the crucial tainted memory region, missing a protocol field. This may happen if the gap in crucial tainted memory region reaches the beginning of the buffer, and contains an extra protocol field not encountered before. Notice however, that when we analyze a heap corruption attack which overwrote control data (a `red` region) on the heap, we will not miss any protocol fields, since the memory dump is performed exactly at the moment of corruption.

**Polymorphism.** By focusing on properties like field length, the signatures are independent of the actual content of the exploit and hence resilient to polymorphism. By focusing on the vulnerabilities, they also detect attacks with 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. As a result, 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 we want to protect. The implication is that *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 may want to characterize 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 protocol 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 in the next section.

Before specifying the signatures, however, we emphasize that making less specific signatures is greatly facilitated when the attack is fingerprinted, i.e., if we know exactly which bytes contributed to the attack. To continue the example, we could simply *try* to see if the overflow also works for

POST request, by crafting a POST message with a similar URL field. We expect much of this process can be automated, although we have not yet attempted to do so.

#### 4.8.2 The final form of *Prospector*'s 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, they should collectively have a length that is less/not less than  $L$ . We can also put some boundaries on given fields, like in the case of heap overflows based on malformed messages. Example signatures can be found in Section 6.1.

#### 4.9 Double-free errors

We added a module to *Prospector* to make it deal with double free attacks. Memory managers are sometimes exploited when a programmer makes the mistake of freeing a pointer that was already freed. Double-free errors do not share the characteristics of heap-corruption attacks in the sense that they do not overflow a buffer, and so when considering the analysis they require special treatment. *Prospector* also contains a module that provides a means of analysis for such attacks.

Double-free exploits may overwrite any location, resembling the complex heap corruption attacks. Similarly, it is highly probable that when a violation is detected, the memory region that was holding the vulnerable buffer is reused and contains unrelated data. To deal with this issue, whenever `free` (or `realloc`) is called, we check for a potential double free error, assuring that the given memory location indeed points to the beginning of an allocated buffer. Otherwise we store the adjacent tainted memory region for possible later use.

To make the module working efficiently, we store a red-black tree of currently allocated memory regions for each address space. It guarantees fast access to the nodes when necessary.

Double free errors do not lead to buffer overflows like the other heap and stack corruption attacks. The current implementation of *Prospector* produces fairly trivial signatures for them by identifying a protocol field which should contain a selected substring of the crucial region. The crucial memory region is determined in the same way as for the complex heap corruption attack. When we pinpoint in the heap dump the address that caused the violation, we take its non-stale tainted neighborhood as the invariable bytes for this attack. Notice that these bytes contain fake heap control data, and so are not supposed to be different in each instance of the message exploiting the vulnerability. However, there is a lot of space for improvement here, and we believe that the accurate data provided by *Prospector* can be used to produce a more powerful signature.

## 5 Implementation details

*Prospector* was implemented for Linux on the *Argos* x86 emulator. In this section we discuss the most important implementation issues.

### 5.1 Monitoring process switches from the hardware

*Prospector* stores information about the allocation and deallocation of stack frames in each process. Thus we need a means to monitor context switches on the level of processor emulator. This is not a trivial problem, as the hardware emulator has no knowledge of processes. The solution for

IA-32 proposed by [14] tracks changes of the `cr3` (or page table base) register, which stores the physical address of the page directory. As a rule, a switch implies changing the set of active page tables, and thus loading `cr3` with the value stored in the descriptor of the process to be executed next. However, the solution is problematic, as Linux avoids this operation in the following cases: (1) when performing a switch between two regular processes that use the same set of page tables, i.e., lightweight processes, and (2) when performing a process switch between a regular process and a kernel thread. Kernel threads do not have their own set of page tables; rather they use the page tables of the regular process that was scheduled last for execution on the CPU.

Proper tracking of context switches proved a very challenging problem. We sketch our solution that is accurate for Linux, so we do not have to worry about missing context switches. In Linux, each execution context that can be independently scheduled has its own process descriptor. Therefore even lightweight processes and kernel threads, have their own `task_struct` and `thread_info` structures. For each process, Linux keeps a memory area, at the beginning of which resides the `thread_info` structure, and the kernel mode process stack grows downward from the end. The length of this memory area is fixed, usually 8K. For reasons of efficiency the kernel stores the 8K memory area in two consecutive page frames with the first page frame aligned to a multiple of  $2^{13}$ . Thus the 19 most significant bits of a memory location inside the kernel mode stack are the address of the `thread_info` structure, which we refer to as  $P$  and serves as a unique process identifier.

Whenever the CPU operates in kernel mode, we can determine  $P$  by taking the 19 most significant bits of the present stack pointer (ESP). As *Qemu*, on which *Argos* is based, translates all guest instructions to host native instructions by dynamically linking blocks of functions that implement the corresponding operations, we can check  $P$  right before the *Argos* emulator in kernel mode executes a block of instructions. On each context switch the OS always executes at least a few instructions in kernel mode, and so we always have a correct value of the process identifier.

## 5.2 Heap Protection

As explained in Section 4, to deal with complex heap corruption attacks, we mark the bytes surrounding allocated chunks of heap memory as **red**. Since we cannot monitor *allocations* at the level of the emulator, we interpose the `malloc` and `free` (also `calloc` and `realloc`) functions in the guest OS, and by means of *argos calls* inform *Qemu* about changes on the heap. *Argos* calls are analogous to system calls in Linux, and are called by trapping with an unused interrupt number (82). Whenever *Argos* receives this interrupt, it passes control to a handler corresponding to the *argos* call number.

When `malloc` returns, we provide *Argos* with the address `addr` and length `len` of the newly allocated chunk. *Argos* marks the bytemap's entry corresponding to `(addr - 1)` as **red**, and also stores the value of `len` as a 'fake' taint value for `(addr - 1)`. This is safe as this address contains non-tainted control information used by the memory allocator and so will never really be tainted. *Argos* makes use of `len` when a memory chunk is released. We also set the red indicator for the byte directly above the chunk of memory. Similarly, when `free` is called, *Argos* cleans the **red** markers adjacent to the released memory buffer.

To support the heap corruption attacks based on malformed messages, we also check whether `len` is tainted. If so, we store its network origins in the red-black tree mentioned in Section 4.9, in a node corresponding to the newly allocated memory chunk.

### 5.3 Prospector tagging

To deal with memory tagging *Argos* introduces a structure similar to page directories in Linux consisting of *pagemaps* and *bytemaps*. A pagemap is an array, where each entry corresponds to a bytemap keeping tags for a particular physical page. Here *Argos* stores all tags on the guest operating system memory, e.g., the network offsets that serve as taint tags. Initially only the pagemap is allocated. Bytemaps are added on demand, when tainted data is copied to a particular physical page for the first time. The network offset tags associated with each byte are 32 bits. To support signature generation we doubled the size of the tag, yielding an additional 32 bits. Of these 32 bits, we designate one bit for the `prev_fresh_taint` and `1st_taint_next_address` indicators, one bit for the red marker denoting critical data on the heap, and the remaining 29 bits for the age stamp. We emphasize that age stamps serve only to compare tainted data, so they need only be incremented if a given value was used as a tag to mark tainted data. As most functions and indeed most processes never touch such data, the age stamp may remain untouched. As a result, the age stamp will wrap much more slowly. We will address age stamp wrapping in Section 5.4.

*Qemu* translates all guest instructions to host native instructions by dynamically linking blocks of functions that implement the corresponding operations. With the aim of tracking tainted data being copied to memory we instrument the `store` function to perform the operations of keeping track of age stamps and setting the extra indicators (`prev_fresh_taint` and `1st_taint_next_address`) described in Section 4.5. Here we also check whether the destination memory location is not marked as `red` (which indicates an overflow and perhaps a complex heap corruption attack, and therefore leads to a dump of the adjacent tainted memory).

### 5.4 Age stamp wrapping

`AgeStamp` is a 29-bit global variable used to draw conclusions about the age in which data coming from the network was copied to a buffer. Thus we wish to avoid problem due to `AgeStamp` wrapping. We could add more bits, but this may not be necessary. We measured the time needed by `AgeStamp` to wrap depending on its length. The tests were performed on the guest OS running `Apache`, receiving 45 requests per second (a rate before it saturates on our emulator). Figure 3 presents the results. In the optimized version we increase `AgeStamp` by 1 only if it was actually used to tag network data in memory. Otherwise the value is updated on each `call` and `ret`.

In the optimized version `AgeStamp` of a guest OS running `Apache` needs almost 16 hours to wrap. We use either of the following solutions to avoid the undesirable scenario described above: (1) restart the honeypot running *Argos* and *Prospector* twice a day, (2) dump all tags when `AgeStamp` wraps. This dump can be used for later analysis and separation of the values from the previous epoch. In the light of the long time needed by the counter to reach the limit both solutions are feasible and we personally prefer the continuous operation of solution 2.

### 5.5 Stale red markers

As mentioned earlier, to handle complex heap corruption attacks, we mark bytes surrounding allocated chunks of memory as `red`. If tainted data is written to a red region, this indicates illegal operations which trigger bookkeeping: the memory region is dumped. As we cannot rely on applications releasing all allocated memory, we may end up with stale red markers, possibly

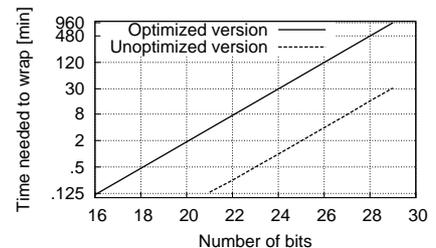


Figure 3: Time to wrap for different sizes of `age_stamp`.

leading to unnecessary dumps of memory regions. We describe here how we solve this problem by removing false red indicators.

First of all, we keep counters indicating the number of red markers associated with each physical page in memory. To deal with the problem in the case of pages for the user stack or kernel memory, we monitor new entries added to the TLB as follows. We keep a table of physical pages associated with the identifier of the last process using it. Whenever a new entry corresponding to a kernel address or the user stack is added to the TLB buffer, we check whether the page has a new owner, and if so, we make sure that it does not contain any red markers. If so, we know that neither the user stack nor kernel memory contains the markers.

For the heap we cannot use this method, since dynamically allocated memory can easily be shared between processes, which could remove our markers. Thus, whenever a new buffer is allocated, we assure that its contents do not contain any red regions. First, we check the counter of red markers associated with the given page (or pages) and, if necessary, *clean* the memory.

## 6 Evaluation

We evaluate *Prospector* along two dimensions: effectiveness and performance. While performance is not critical for a honeypot, it needs to be fast enough to generate signatures in a timely fashion.

### 6.1 Effectiveness

To test our analysis and signature generation, we launched a number of real attacks (as well as hand-crafted ones) against Linux on top of *Argos*. We have not experimented with Microsoft Windows since a small part of the functionality in *Prospector* is OS-specific, i.e., `malloc` and `free` function interposition and (partly) process switch monitoring. For launching attacks, we used the Metasploit framework<sup>1</sup> and Milw0rm<sup>2</sup>. In this section we illustrate how *Prospector* deals with two representative stack- and two heap overflow attacks.

**PeerCast Stack Overflow.** A remote overflow exists in PeerCast v0.1216 and earlier [31]. It fails to perform correct bounds checks on parameters passed in a URL, resulting in a stack-based overflow. An overly long query overwrites EIP stored on the stack. When the function returns, *Argos* raises an alert. Our analysis engine correctly separated stale data on the stack. It encountered a 4-byte discontinuity in the critical tainted memory region, skipped it, and continued down the memory buffer. The final signature for this attack contains the following fields:

```
(application: PeerCast, version: v0.1212, (type: value_field, name: method, value: GET),  
(type: critical_field, name: query), (type: critical_length, value: 476)).
```

**Subversion Stack Overflow.** There is a remote overflow in Subversion 1.0.2 [24] which fails to bounds check when calling `sscanf()` to decode old-styled date strings. By sending a crafted request via the `get-dated-rev` svn command, a remote attacker can cause a buffer overflow. In our experiment, an overly long `week day` overwrites EIP stored on the stack. As in the previous exploit, *Argos* raises an alert, and analysis starts. This time no discontinuities on the stack were encountered. The resulting signature contains the following fields:

```
(application: Subversion, version: 1.0.2, (type: value_field, name: command, value: get-dated-rev),  
(type: critical_field, name: week_day), (type: critical_length, value: 20)).
```

---

<sup>1</sup>The Metasploit Project, <http://www.metasploit.com>.

<sup>2</sup>Milw0rm, [www.milw0rm.com](http://www.milw0rm.com)

**WvTftp Heap Overflow.** A heap-based overflow in the WvTftp 0.9 allows remote attackers to execute arbitrary code via a long option string in a TFTP packet [28]. The option name value pairs are given as a NULL terminated option name, followed by an ascii representation of the number value. The function `atoi()` is used on the value string, and as long as the original part of the string equals a value  $> 8$  and  $< 65464$ , the string is `strcpy`'d into the heap buffer. By supplying a long string for the value, the buffer can be overflowed. *Argos* correctly noticed that the heap control `red` region was overwritten with network data. In the analysis phase, no discontinuities on the heap were encountered. The resulting signature contains the following fields:

```
(application: WvTFTP, version: 0.9, (type: value_field name: Opcode, value: Read Request (1)),
(type: critical_field, name: Blocksize option), (type: critical_length, value: 557)).
```

**Asterisk Heap Overflow.** Asterisk<sup>3</sup> is a popular software telephony server. The Asterisk Skinny channel driver for Cisco SCCP phones in v1.0.11 and earlier, v1.2.12 and earlier (`chan_skinny.so`) incorrectly validates a length value in the packet header. An integer wrap-around leads to a heap overwrite, and arbitrary remote code execution [29]. Asterisk checks whether the inequality  $(\text{length\_value} + 8 \leq 1000)$  holds to convince itself that the user-supplied message fits in the local buffer of size 1000. Because of the integer wrap, the result of the comparison is positive. And then, the 4 bytes length are copied to the vulnerable buffer, and a `read` operation is performed storing  $(\text{length\_value} + 4)$  bytes of the rest of the message on the heap. The attack benefits from the fact that `read` takes an unsigned value as the last parameter, so `length_value` is interpreted as a very large number instructing `read` to write beyond the allocated 1000 byte length of the vulnerable buffer. *Argos* detects that the control `red` region on the heap gets overwritten with network data, and dumps the corresponding memory area. In the analysis phase, we first come across the whole SKINNY message but the length field (this part has the same age stamp). Next, we include the 4 bytes underneath it, forming the length, in the crucial tainted memory region (since it is a tainted region with correctly fitting age stamps). Thus the signature specifies the whole SKINNY Packet for Asterisk 1.0.10 not to exceed 1000 bytes. Notice, that even though the length field does not need to be included in the signature, the attack description is still absolutely correct.

## 6.2 Performance

For realistic performance measurements we compare the speed of code running on *Argos* and *Prospector* with that of code running without emulation. Note that while this is an honest way of showing the slowdown incurred by our system, it is not necessarily the most relevant measure. After all, we do not use *Prospector* as a desktop machine and in practice hardly care whether results appear much less quickly than they would without emulation. The only moment when slowdown becomes an issue is when attackers decide to shun slow hosts, because it might be a honeypot. To the best of our knowledge, automated versions of such attacks do not exist in practice.

Performance evaluation was carried out by comparing the observed slowdown at guests running on top of various configurations of *Prospector* and unmodified *Argos* with the original host. The host used during these experiments was an Intel(R) Xeon(TM) CPU at 2.8GHz with 2048KB of L2 cache, and 4GB of RAM, running Gentoo Linux with kernel 2.6.15.4. The guest OS ran Ubuntu Linux 5.05 with kernel 2.6.12.9, on top of *Qemu* 0.8, *Argos* and *Prospector*. To quantify the observed slowdown we used *Apache* 2.2.3. We chose *Apache* because it is a popular web server and thus it enables us to test the performance of a network service (a domain for which *Argos* was designed). We measured its throughput in terms of processed requests per second and the

---

<sup>3</sup>[www.asterisk.org](http://www.asterisk.org)

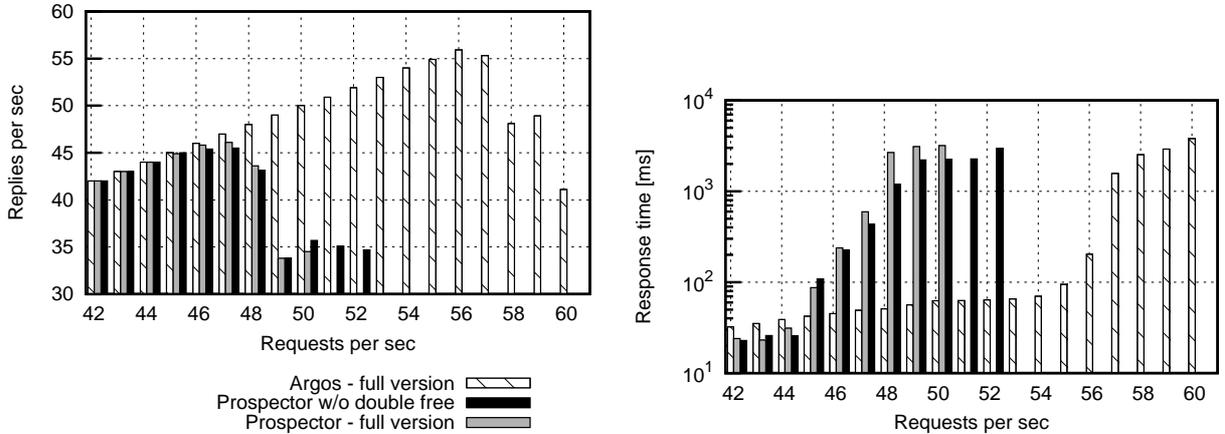


Figure 4: Apache throughput in terms of maximum processed requests per second, and the average response time.

corresponding average response time. We used `httperf` for generating requests. `Httpperf` is able to generate high rates of single-file requests to determine a web server’s maximum capacity.

Figure 4 shows the results of the evaluation. We tested the benchmark application at the guest running over *Argos*, and two different configurations of *Prospector*: both with and without the double free extension module. The graphs show that the achieved throughput increases linearly with the offered load until the server saturates at a load of 48 calls per second in the case of *Prospector* and 57 for *Argos*. This is also reflected in the figure presenting response times. The graph shows that the response time starts out at about 20-30ms, and then gradually increases until the server becomes saturated. Beyond this point, response time for successful calls remains largely constant at 3000ms.

Notice that there is no difference in performance between the two versions of *Prospector*. Calls to memory management related functions are rare in the context of the whole web server application, and so additional harmless operations on each `malloc()` and `free()` appear not to decrease performance.

We can conclude that the overhead expressed in throughput of a web server incurred by *Prospector* compared to *Argos* is approximately 16%. We have also performed measurements of slowdown in comparison with the original host (refer to [26] for the full performance evaluation of *Argos*.) *Apache* on *Argos* is about 15 times slower than the one run on the native operating system (on *Prospector* 18 times). We emphasize that we have not used any of the optimization modules available for *Qemu*. These modules speed up the emulator to a performance of roughly half that of the native system. While it is likely that we will not quite achieve an equally large speed-up, we are confident that much optimization is still possible. Moreover, even though the performance penalty is large, personal experience with *Argos* and *Prospector* has shown us that it is tolerable.

## 7 Conclusions

We have described *Prospector*, an emulator capable of tracking which bytes contribute to an overflow attack on the heap or stack. By careful analysis, and keeping track of the age of data, we manage to provide such information with greater accuracy than previous approaches while maintaining reasonable performance. The information is important for security experts. We have also used the information to generate signatures for polymorphic attacks by looking at the length of protocol

fields, rather than the actual contents. In practice, the number of false positives for the signatures is negligible and the number of false negatives is also low. At the same time, the signatures allow for efficient filters.

## Acknowledgment

This research was supported by the EU FP6 NOAH project.

## References

- [1] P. Akritidis, E. P. Markatos, M. Polychronakis, and K. D. Anagnostakis. Stride: Polymorphic sled detection through instruction sequence analysis. In *Proceedings of the 20th IFIP/SEC 2005*, 2005.
- [2] Anonymous. Once upon a free(). <http://doc.bughunter.net/buffer-overflow/free.html>.
- [3] W. X. S. Bhatkar and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of 15th USENIX Security Symposium*, 2006.
- [4] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, May 2006.
- [5] R. Chinchani and E. Berg. fast static analysis approach to detect exploit code inside network flows. In *In Recent Advances in Intrusion Detection*, Seattle, WA, September 2005.
- [6] M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant. Semantics-aware malware detection. In *Security and Privacy Conference*, Oakland, CA, May 2005.
- [7] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of Internet worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, October 2005.
- [8] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, PerryWagle and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *7th USENIX Security Symposium*, 2002.
- [9] H. Dreger, A. Feldmann, M. Mai, V. Paxson, and R. Sommer. Dynamic application-layer protocol analysis for network intrusion detection. In *Proceedings of 15th USENIX Security Symposium*, 2006.
- [10] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, and B. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *Proceedings the IEEE Symposium on Security and Privacy*, Oakland, CA, 2004.
- [11] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *Proceedings of the IEEE Security and Privacy Conference*, Oakland, CA, 2003.
- [12] P. Fogla and W. Lee. Evading network anomaly detection systems: formal reasoning and practical techniques. In *Proceedings of the 13th ACM CCS*, pages 59–68, New York, NY, USA, 2006. ACM Press.
- [13] J. T. Giffin, S. Jha, and B. P. Miller. Automated discovery of mimicry attacks. In D. Zamboni and C. Krügel, editors, *RAID*, volume 4219 of *Lecture Notes in Computer Science*, pages 41–60. Springer, 2006.
- [14] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *Proceedings of the USENIX 2006*, Boston, MA, June 2006.
- [15] C. Krügel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *14th Usenix Security Symposium*, Baltimore, MD, August 2005.
- [16] C. Krügel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *RAID*, Seattle, WA, October 2005.
- [17] C. Krügel, T. Toth, and E. Kirda. Service specific anomaly detection for network intrusion detection. In *Proceedings of the 2002 ACM symposium on Applied computing*, pages 201–208. ACM Press, 2002.
- [18] C. Krügel and G. Vigna. Anomaly detection of web-based attacks. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 251–261, New York, NY, USA, 2003. ACM Press.
- [19] Ktwo. Admmutate: Shellcode mutation engine. <http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz>, 2001.
- [20] Z. Liang and R. Sekar. Fast and automated generation of attack signatures: a basis for building self-protecting servers. In *Proceedings of the 12th ACM conference on Computer and communications security*, 2005.

- [21] M. V. Mahoney. Network traffic anomaly detection based on packet bytes. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 346–350, New York, NY, USA, 2003. ACM Press.
- [22] J. Newsome, B. Karp, and D. X. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *IEEE Symposium on Security and Privacy*, May 2005.
- [23] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the NDSS*, 2005.
- [24] Open Source Vulnerability Database. Osvdb id: 6301, subversion date parsing overflow. <http://osvdb.org/displayvuln.php?osvdbid=6301>, 2004.
- [25] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. Sharif. Misleading worm signature generators using deliberate noise injection. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, May 2006.
- [26] 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.
- [27] A. Rahbar. Stack overflow on windows vista. White Paper, Sysdream, [http://www.sysdream.com/article.php?story\\_id=241&section\\_id=77](http://www.sysdream.com/article.php?story_id=241&section_id=77), July 2006.
- [28] Secunia. Cve-2004-1636 wvftft buffer overflow vulnerability, October 2004.
- [29] SecuriTeam. Asterisk skinny unauthenticated heap overflow, October 2006.
- [30] SecurityFocus. Can-2003-0245 apache apr-psprintf memory corruption vulnerability. <http://www.securityfocus.com/bid/7723/discussion/>, 2003.
- [31] SecurityFocus. Cve-2006-1148 peercast.org peercast remote buffer overflow vulnerability. <http://www.securityfocus.com/bid/17040/info>, 2006.
- [32] T. Toth and C. Krügel. Accurate buffer overflow detection via abstract payload execution. In *Recent Advances in Intrusion Detection, 5th International Symposium*, 2002.
- [33] US-CERT. Vulnerability notes database. <http://www.us-cert.gov>, 2007.
- [34] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: vulnerability-driven network filters for preventing known vulnerability exploits. *SIGCOMM Comput. Commun. Rev.*, 34(4):193–204, 2004.
- [35] K. Wang, G. Cretu, and S. J. Stolfo. Anomalous payload-based worm detection and signature generation. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection*, 2005.
- [36] X. Wang, Z. Li, J. Xu, M. K. Reiter, C. Kil, and J. Y. Choi. Packet vaccine: black-box exploit detection and signature generation. In *Proceedings of the 13th ACM CCS*, 2006.
- [37] X. Wang, C.-C. Pan, P. Liu, and S. Zh. Sigfree: A signature-free buffer overflow attack blocker. In *Proceedings of 15th USENIX Security Symposium*, 2006.