Online Signature Generation for Windows Systems

Lixin Li and James E. Just
Global InfoTek, Inc.
Reston, VA, USA.

R. Sekar
Stony Brook University
Stony Brook, NY, USA.

***Abstract*—In this paper, we present a new, light-weight approach for generating filters for blocking buffer overflow attacks on Microsoft Windows systems. It is designed to be deployable as an "always on" component on production systems. To achieve this goal, it avoids expensive and intrusive techniques such as taint-tracking. The online nature of our system enables it to provide protection from a range of memory corruption exploits, including those involving unknown vulnerabilities, or known vulnerabilities but unknown exploits. In contrast, most previous signature generation techniques need to be run in sandboxed environments, and need working exploits to generate signatures. Moreover, our technique overcomes the "gap" problem faced by previous signature generation mechanisms, i.e., when the vulnerable memory region is corrupted between the overflow and the time an attack is detected. Another novel feature of our approach is that it is able to reason about likely lengths of vulnerable buffers, which can lead to more accurate signatures. Our experimental results are very promising, and demonstrate that the approach can generate effective signatures for many synthetic and real-world vulnerabilities.**

***Keywords*-signature generation; buffer overflow; self-healing**

I. INTRODUCTION

Buffer overflows continue to be one of the most common vulnerabilities prevalent today, especially dominating among "critical updates" from vendors such as Microsoft. This factor has motivated the deployment of defenses such as address space randomization (ASR). Unfortunately, these defenses simply convert a working exploit into a crash. Since server restarts typically take significant time (of the order of a second or more), an attacker can bring down a server by repeatedly attacking a server with just a few packets every second. Indeed, with the advent of botnets and widespread prevalence of cyber extortion based on DDoS threats, such attacks become very easy to carry out, yet very hard to defend against. Moreover, repeated attacks can compromise randomization defenses, especially when the range of randomization is small (e.g., 256 in the case of Windows Vista).

The drawbacks of existing buffer overflow defenses have motivated the development of *automated signature generation* techniques that generate filters to block attack-bearing inputs from being delivered to a vulnerable server. Initially, this line of research targeted network worms, and relied on "content-based signatures," where the signature captured characteristics of the attack payload. Unfortunately, polymorphic attacks can evade these signatures, and hence subsequent research focused on "vulnerability-oriented signatures" [10], [5], [2], [7], [21]. These signatures capture the characteristics of the underlying vulnerability (e.g., maximum length of input) rather than those of the attack payload. Recent trends in this area has tended to emphasize signature generality: i.e., its ability to block a wide a range of exploit. This trend has favored the development of heavy-weight signature generation techniques [5], [4], [2], [7], [21] that were best suited for offline operation on honeypots or a sandbox. However, offline techniques have some drawbacks:

- *Defenses for zero-day exploits.* The protection offered by techniques deployed on honeypot or sandboxed systems will be unavailable to a production server until the vulnerability is attacked on one of the honeypot systems on which the signature generator is deployed, and the resulting signature distributed. This leads to unnecessary delays in signature deployment, leaving critical servers exposed to attacks. Worse, many of the previous techniques rely on generating variants of attacks and replaying them, and this would typically require the availability of working exploits. Unfortunately, working exploits won't be available for unknown vulnerabilities, as well as for many known vulnerabilities.
- *Defense against guessing attacks.* Since guessing attacks on ASR can be completed within a short period — say, several seconds — a production server may be compromised before a signature is generated by a heavy-weight technique; or between the time it is generated and widely distributed.

In contrast, we present a light-weight technique that is suitable for on-line operation on production systems. Our technique differs from COVERS [10], another online signature generation technique, in many important ways. First, the focus of this paper is on Microsoft Windows, while COVERS is based on Linux. Second, our work makes innovative use of modern memory error defenses (such as return address cookies, heap metadata cookies, SafeSEH protection, data execution prevention) to improve signature generation. Finally, our work develops better analysis and correlation techniques that enable robust signatures to be

This work was funded in part by Defense Advanced Research Project Agency (DARPA) under contract N00178-07-C-2005. Sekar's work was also supported by AFOSR grant FA9550-09-1-0539, ONR grant N000140710928 and NSF grants CNS-0627687 and CNS-0831298. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, the Naval Surface Weapons Center, AFOSR, ONR, NSF, or the U.S. Government.

generated in scenarios where COVERS may fail.

A. Approach Overview and Contributions

Our primary goal is to defend against zero day attacks in production environments. Our focus is on light-weight analysis techniques that can be invoked on-demand, so that zero-day attacks can be detected and quickly analyzed as they happen, while avoiding any significant overheads during normal operation. Our approach leverages (fail-crash) memory corruption defenses built into modern operating systems for detecting memory corruption attacks. It then uses an analysis of victim process memory to identify possible memory corruption targets, and correlates the data surrounding the target with recent inputs to identify the input(s) responsible for the attack. It then generates a signature (also called a blocking filter or simply a filter) that characterizes this attack-bearing input. This signature is then deployed to filter out repetitions of the exploit and (hopefully) its variants.

Our approach is based on the observation that most memory corruption attacks involve copying data that overflows a buffer and overwrites an adjacent pointer. If this pointer value is incorrect (as is typically the case in the presence of ASR), a memory exception is triggered at the time this pointer value is dereferenced. This exception initiates an analysis phase that involves correlating the crash-causing data, namely, the data in the destination buffer that overflowed, to an input. Previous work has typically relied on taint tracking [13] to identify this input, but unfortunately, taint-tracking on C/C++ programs introduces high runtime overheads, slowing down programs by a factor of two to ten.

We observe that most memory corruption attacks involve simple copying of data. This raises the possibility that correlation can be made by simply comparing the crash-causing buffer to inputs, as done in the COVERS [10] work. However, this approach runs into problems if the copied contents get corrupted after the copy operation. For instance, consider a stack-resident buffer. A buffer overflow will typically corrupt a few local variables on the stack and then the return address. If these local variables change before the return (i.e., the time of crash), the buffer contents may not exactly match the input. On our experimental platform, namely, Windows, we found that such changes to inputs are common, and hence an exact matching algorithm can fail to correlate corrupted buffer with recent input.

We made the key observation that the sort of input changes that occur due to program execution can be accommodated using an approximate rather than an exact matching algorithm. Based on this observation, we made use of an approximate substring searching algorithm for performing input correlation. After performing correlation, our technique generates a blocking filter that characterizes the attack input without matching any benign inputs. Like previous techniques, the signature generation phase is guided by input format specifications.

Although a sophisticated classification technique could be used for generating signatures from a list of benign and attack-bearing inputs, our current focus is on demonstrating the feasibility of our approach, and hence we have focused on simple techniques that reason about input lengths and other characteristics that typically differ between benign inputs and inputs responsible for crash.[1]

- Instead of tracking every move of input at machine instruction level, as is done with many previous approaches that relied on taint-tracking, we are able to infer flow of data from input to a vulnerable buffer by using an efficient post-crash analysis of victim memory.
- Our techniques take advantage of modern (fail-crash) memory exploit defenses built into Windows. Many of these defenses (e.g., return address cookies) detect memory corruptions at an earlier stage than more generic defenses such as ASR. Our techniques leverage this factor to develop simpler and more robust mechanisms for identifying key details of the exploit.
- A key insight in our approach is that the use of an approximate (rather than an exact) string matching algorithm can lead to a technique that can be robust in the face of "gaps" that occur due to memory updates (or corruptions) that may occur between the time of buffer overflow and the time of crash. Such gaps were the norm rather than the exception on our experimental platform.
- Unlike previous light-weight techniques that relied purely on learning to develop length-based signatures, we develop a new technique that can infer the likely lengths of vulnerable buffers using a runtime memory analysis and string matching. This enables our approach to generate more accurate signatures.
- Our system can mitigate Denial-Of-Service attacks as well as zero-day working exploits on memory error vulnerabilities. This contrasts with other light-weight signature generation techniques that generate blocking filters based on checking if the "jump addresses" incorporated in the attack has a certain value (or range of values), and hence can be fooled by DoS attacks that need not contain a valid jump address.
- We have implemented the technique and evaluated it on a synthetic server seeded with a variety of vulnerabilities, as well as a number of real world applications. Our experiments show that our technique is effective

[1] Naturally, this focus implies that certain types of overflow vulnerabilities won't be handled, such as those where the overflow involves inconsistencies between multiple message fields (e.g., a field that specifies the length of a string in another message field, but the string happens to be longer than the value specified in the first field) or concatenation of multiple message fields. Nevertheless, our techniques were able to successfully generate signatures for most real-world and synthetic vulnerabilities considered in our experimental evaluation.

and has low performance overheads that is acceptable for production use.

II. Impact of Windows Exploit Defenses

Windows Vista provides the following defense mechanisms against memory corruption exploits:

- */GS option ("return cookies"):* This is a compiler option that generates additional code to detect return address (and saved base pointer) corruption. Similar in style to StackGuard [6], the runtime check involves verifying the integrity of a cookie that resides before the protected items on the stack.
- */SafeSEH option:* This is another compile-time option that produces a table of safe Structured Exception Handlers (SEH). Before control is transferred to any exception handler code at runtime, the target address must be present in this table. A runtime SEH link list validation is built in Windows Server 2008.
- *Heap overflow protection:* Traditional heap overflows involve corruption of heap metadata. Modern versions of Windows incorporate cookies and other mechanisms to detect metadata overwrites and abort the program.
- *Address-space randomization:* Windows Vista incorporates ASR, but the range of randomization is just 256. This means that individual exploits will be blocked with a probability of 99.6%, but brute force attacks have a 50% chance of succeeding in 128 attempts.
- *Data Execution Prevention (DEP):* DEP prevents injected malicious code from executing in non-executable (NX) memory regions such as the stack or heap.

These protections, collectively referred to as modern protections, seem to provide a formidable defense at first glance. However, there are a number of reasons why memory corruption attacks continue to be a serious threat:

- There still exist a vast number of systems that are running older versions of Windows (e.g., XP) on which some of these protections are unavailable.
- Some of these defenses (in particular, /GS, /SafeSEH, NX, and some aspects of ASR) require a software vendor to opt-in. Some vendors, due to compatibility or performance concerns, don't enable these protections.
- Most of these protection mechanisms can be circumvented using specially crafted attacks. For instance, return address cookies can be bypassed by corrupting data or function pointers on the stack; derandomization attacks [16] are easy to mount on Windows ASR due to its small range of randomization; return-to-libc attacks, and the more general technique of return-oriented programming [15], overcome NX. Finally, non-control data attacks [3] can defeat almost all of these defenses.
- Finally, even when effective, these defenses convert working exploits into crashes, which are seldom acceptable on a production server. Worse, attackers can utilize these crashes to mount targeted DoS attacks.

As a result of these factors, memory corruption exploits remain popular on Windows. But the defense mechanisms have certainly skewed the distribution of exploits to some extent. For instance, traditional heap overflows are rare on modern Windows systems due to the effectiveness of heap overflow protections. Traditional stack-smashing remains possible on systems unprotected by /GS, but attacks that corrupt exception handler pointers tend to be more popular, as they work more reliably. We leverage these facts in this paper: we develop general techniques that can perform attack data correlation to input, while leveraging specific aspects of Windows exploits to improve signature generation speed.

III. Technical Approach

Remote memory exploits rely on the ability of an attacker to control the input to a vulnerable program. Taint-based techniques, which track the flow of remote (untrusted) data within the target program, have proven to be very effective in detecting these attacks, as well as in correlating attacks to specific inputs that contained the attack. Unfortunately, taint-based techniques incur high performance overhead. Moreover, they require deep instrumentation, wherein every instruction in the original program is augmented with additional instructions to perform taint-related computation. Our approach therefore relies on inferring taint by comparing memory contents with recent inputs.

We made similar observations in the context of injection attacks on web applications, leading to the development of a taint inference [14] approach. However, web application vulnerabilities are quite different from memory corruption vulnerabilities. For instance, they involve string data in almost every case. More important, the technique described in [14] is able to examine suspicious data before it is used in an injection attack, whereas with memory corruption, attacks are detected some time after data overflow takes place. Since these attacks involve memory corruption, the copied data may in turn be corrupted before it is analyzed by our post-crash analyzer. For instance, if the vulnerability involves a stack resident buffer, subsequent program execution may result in changes to the data beyond the end of the buffer, which may store local variables or saved registers. Thus, we needed to develop a technique that infers taint in spite of possible corruption of some sections of data copied into the vulnerable buffer.

This problem of partial data corruption has been pointed out by others, e.g., as the "gap" problem in [20]. Contrary to what was initially thought of as a rare case, we encountered gaps much more frequently on our experimental platform (Windows), and hence believe that such corruptions may be the norm rather than the exception.

To address corruption of overflowed data, we rely on approximate matching rather than exact matching for inferring taint (Approximate string matching is also used in [14], but for a different purpose: to deal with minor input

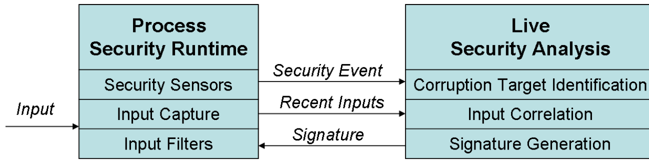

Figure 1. System Overview

transformations that take place in web applications, e.g., conversion of lowercase characters to uppercase, replacement of spaces with underscore, etc.). This technique provides a general solution to the problem of gaps, without any regard to the specifics of the objects involved in corruption, i.e., it does not matter if they relate to local variables, compiler-generated temporaries, global variables, heap data, etc. Another innovative aspect of our approach is that we use this corruption to our benefit: *the byte positions where corruption has occurred provide clues as to where the end of the buffer is, and hence gives us a handle on the size limits that should be imposed on the input to avoid the overflow.*

A. System Overview

Figure 1 illustrates our system. On the left side is the Process Security Runtime (PSR) hosted inside the protected application process. PSR is responsible for capturing inputs, usually by intercepting network or file reads, enforcing input filtering when applicable, and emitting security events from security sensors to Live Security Analysis (LSA) on the right side. LSA can be deployed at the local host or at a remote host. It remains dormant during execution and will be invoked only when a security event is received from PSR. LSA performs security analysis, input correlation and generates input filter when applicable. The rest of the components in the figure are described below.

Security sensors are responsible for emitting security events of interest to LSA. The most basic form of a security event is a memory access exception, which is raised when an instruction references an invalid address. In addition, protections such as /GS and /SafeSEH can serve as sensors as well, enabling attacks to be detected somewhat earlier.

Input Capture. When an input is received, our system will make a copy and keep recent inputs around. Because attacks typically lead to a crash quickly, these inputs are not buffered for very long. The number of inputs to be buffered, as well as the maximum time duration for which they may be buffered, are both configurable.

Input Filter. When new inputs come in, they are matched against the existing list of signatures previously generated. Any input matching a signature will be dropped[2]. When inputs received on a TCP connection are dropped, the connection is also severed so as to preserve (reliable data delivery) semantics of TCP. When the application tries to read a dropped input, a network error is signaled by the input filter. Most applications expect occasional network errors, and contain error-handling code to recover from this error and continue normal operation.

Input filters can support a "logging" mode for specified signatures, instead of the default "blocking" mode. This mode is useful if the confidence level in a signature is not high enough. In order to generate more accurate signatures, additional logging is turned on when input matching a "logging" signature is received. This additional on-demand logging can be very helpful because it can potentially provide intermediate states before a crash, and hence help us construct the traces of events leading to the crash. It is also possible for the level of logging to be escalated gradually. For instance, in the first step of escalation, additional function interceptors could be turned on and the inputs/outputs of these functions recorded. In the last step of escalation, an entire memory dump could be produced.

Corruption Target Identification (CTI) is the first step initiated when a security event is received by the live security analysis (LSA) component. As described in Section III-B, CTI computes a list of candidate memory locations that have been targeted in the just-detected exploit.

Input Correlation is concerned with identifying the input that was copied to the corruption target, and pinpoint the bytes of this input involved in the attack. Our input correlation relies on a fast approximate substring matching algorithm and is described in more detail in Section III-C.

Signature generation is the last step of our analysis. Our focus in this paper is on the correlation and corruption target identification steps, but not on the development of sophisticated signature generation algorithms. This limits the class of memory corruption exploits that our current prototype can handle, but on the other hand, the online nature of our signature generator gives us more flexibility. For instance, a coarse signature could be generated, and refined over time. Additional logging could be initiated during this to help guide this refinement. Signature generation is further described in Section III-D.

B. Corruption Target Identification (CTI)

The goal of the CTI phase is to identify candidate locations that were corrupted in the exploit. CTI may return multiple targets, and may, in the worse case, need to scan the entire victim process memory to complete its task. Our implementation performs this search in several steps, starting from the most specific (and efficient) step and progressing to the most general (and expensive) step. The search is stopped at the first step where signature generation succeeds.

/GS cookie corruption. When this corruption is detected, Windows Vista throws an `int 0x3` exception with STATUS_STACK_BUFFER_OVERRUN debugging message.

[2] Since our signatures rely on length rather than content, all attacks that exploit the same underlying vulnerability can be blocked, without any regard to their payloads.

With older Windows versions, our system can set a breakpoint in `kernel32!UnhandledExceptionFilter` to detect whether a security cookie got tampered with. At this point, we can easily obtain the location of the corrupted cookie, and this is returned as the corruption target.

/SafeSEH corruption. On Windows, exception handlers are maintained as a linked list on stack. A pointer to the first entry in this list is stored in the Exception Registration Record in the Thread Environment Block, which in turn resides at `FS:[0]`. CTI traverses this linked list to identify the location of corrupted SEH entry. This location is returned as the corruption target.

Heap metadata corruption detection. In this case, the location of the corrupted heap block can be obtained and returned as the corruption target. However, we have not implemented this step in our prototype since it requires a different implementation for Windows versions that support heap metadata protection and those that don't. Instead, our approach was to "wrap" calls to malloc-related functions to keep track of allocated heap blocks and their lengths. This information is utilized, as described in step (4) below, to identify corruption targets when an ASR/NX-induced memory access violation occurs after heap metadata corruption.

Generic ASR or NX detection. In the presence of NX, injected code attacks always lead to a memory access violation. In the case of ASR, attack detection is probabilistic: there is of course a small probability that the attacker is able to guess the correct address, and the attack goes undetected. Naturally, no signature generation is possible in this case. A more likely scenario is that the attacker does not guess the right address, but manages to guess a valid address. In this case, a memory access violation may not happen immediately; if and when it happens, it may be too late to correctly diagnose the problem. As observed by Liang et al [10], most servers use only a fraction of available memory space, and hence the probability of delayed access violation is quite small. We accept that signature generation would fail in such cases, and will need to be attempted the next time the attack is repeated[3]. Thus, we limit our discussion to cases where there is an immediate access violation.

When a memory access violation occurs, we can easily obtain the *faulting address*, i.e., the memory address whose access caused the violation. However, in a typical memory exploit based on corrupting a pointer value, the faulting address corresponds to the *value stored at the corruption target*, and not the corruption target itself. We therefore use the following steps to identify the corruption target:

1) CTI examines the memory locations within a few bytes of ESP to check if they contain the faulting address. (This step will succeed on code that is not compiled with /GS option.)
2) CTI traverses the SEH list to check if (a) any of exception handlers point outside legitimate code sections, and (b) if any of the links in the list are broken. In either case, the location of the corresponding SEH entry is returned as the corruption target. (This step will succeed for SEH-based exploits when the /SafeSEH option is not used.)
3) CTI traverses the entire stack of the faulting thread, examining the stack for locations that contain the faulting address, and returning all those locations as corruption targets. (This step succeeds when the corruption target is on the stack, but the previous two cases don't apply.)
4) CTI traverses the heap blocks to identify if any of their metadata fields contain the faulting address. All such locations are returned as the corruption targets. (This step succeeds in the case of heap overflows.)
5) Finally, if all previous steps fail, CTI searches the entire data section for locations containing the faulting address, and return all of the matching locations as corruption targets.

We remark that in our search, rather than looking for occurrences of the faulting address, we look for values that are close (within a range of ± 16) to the faulting address.

C. Input Correlation

Once the initial analysis is done, LSA has a list of *candidate* corrupted targets. Whether they are really the targets would be determined by comparing the content surrounding the target with recent inputs. We use approximate matching to cope with the "gap" problem outlined earlier[4]. We want to demonstrate not only that we can handle gaps, but also that by proactively identifying the gap, we can use it to enhance the accuracy of the vulnerability/signature because local variables are overflowed before the usually sensitive targets like return address, exception handler etc.

Before input correlation proceeds, recent inputs are first parsed and broken into message fields. Specifically, they are broken into $\langle name, value\rangle$ pairs[5]. Then the following steps are used to process each candidate C returned by the CTI.

- *Quick elimination.* The correlator scans the list of all $\langle name, value\rangle$ pairs and eliminates those not containing the value stored at C. This step speeds up correlation (by avoiding the next few steps for most benign inputs), but has no impact on the signature that is ultimately generated.

[3] If a server uses 10% of its address space, then the expected number of attacks for an immediate crash to occur can be calculated as 1.1.

[4] As noted in [20], the gap problem poses a challenge for taint-tracking techniques as well – it is not unique to taint-inference.

[5] Accurate parsing of complex network protocols can require significant effort. There is obviously a trade-off between this effort and the ability to generate accurate signatures. Since protocol parsing is not the main focus of this work, we rely on approximate parsing: i.e., breaking the message into certain key fields, while accepting the possibility that some complex substructures may not be broken into individual fields.

- *Approximate substring match.* This step is undertaken for each $\langle name, value \rangle$ pair that remains after the previous step. Let s denote the content of memory locations around the corruption target C, specifically, locations $C - k_1 * |value|$ through $C + k_2 * |value|$, where $|value|$ denotes the length of $value$; and k_1 and k_2 are tunable constants between 0 and 2. We use approximate substring match to identify, among all substrings of s, those substrings u of s that have the smallest edit distance to $value$. In this regard, we limit ourselves to substrings that are within an edit distance threshold d that is specified[6]. Reference [14] describes an algorithm speeding up approximate string matching so that it provides much better than quadratic time performance under these conditions.

Once a matching substring u is identified, we check if the approximate string match reports a cluster of "delete" operations in matching $value$ with u, and if this cluster is located close to (and before) C. If so, the beginning of these delete operations is identified as the beginning of the gap.

The outputs of input correlation step include the message and field involved in the attack, the corresponding location of u and C, and the location of a gap, if any. If multiple candidates succeed in the approximate match step, we select the best candidates (based on the length of match and edit distance) and only output those.

There are times when there are a number of candidates $C_1 < C_2 < \cdots < C_n$ that are close to each other, e.g., when an attacker replicates the jump address many times. Rather than running the approximate substring search step n times on these candidates, we can simply run it once between locations $C_1 - k_1 * |value|$ through $C_n + k_2 * |value|$. Note, however, that the CTI step won't return a sequence of locations just because a jump address is replicated: in particular, in the case of /GS or /SafeSEH protections, as well as heap overflows, our CTI step will be able to pinpoint the candidates since it has accurate information on the locations of security-sensitive targets.

D. Signature Generation

Our focus in this work is on length-based signatures. They take the form $\{$`Name=`$name$`, MAXLEN=`$l\}$, where $name$ is the name of a message field, and l specifies the maximum length of this field. Unlike previous approaches such as COVERS [10], our technique is able to set a maximum length without having to rely entirely on training data. However, if there are multiple candidates returned by the input correlation step — a situation that we have not encountered in our experiments — then we can select the one that yields the best signature in terms of the separation between the lengths of benign (i.e., training) inputs and malicious input(s).

[6]In our experiments, we set d to be 0.2, which roughly means that there can be at most 20% difference between two strings.

The value of `MAXLEN` is determined as follows. If a gap is present, then the maximum length of field $name$ is set as the distance between the beginning of u and the beginning of the gap. A gap usually occurs because some variable that resides past the end of a vulnerable buffer was modified. Thus the beginning of the gap serves as a good indicator of where the vulnerable buffer ends.

When no gap is present, we treat C (the corruption target) as an upper bound, and set `MAXLEN` as the distance between the beginning of u and C. This makes sense because C denotes the location of a security-sensitive target that was corrupted by the attack, and this target should reside past the end of the buffer. In many cases, including heap metadata corruption and /GS cookie and /SafeSEH corruption, the CTI step provides an accurate value for C, and hence there is a high confidence in `MAXLEN`. But in some cases, we do not have sufficient confidence in C, e.g., when the entire stack is scanned for occurrences of a faulting address, and this address is repeated many times in consecutive memory locations L_1 through L_n. In such a case, to avoid false positives, we should use L_n to set the maximum length.

There may be instances where the signature generated as described in the above paragraph may not be satisfactory. In particular, the maximum length l computed may be too long or too short. The former case leads to false negatives, i.e., an input that satisfies the length constraint still leads to an exploit. The online nature of our technique enables tuning of the length to address this problem. In particular, when false negatives are encountered, we can adjust the maximum length to be the geometric mean of b_{max} and l, where b_{max} denotes the maximum length of this field among benign inputs. It is possible that this new value of l is still too long, and needs to be further refined. The use of geometric mean limits the number of such refinement steps needed to $\log(m_{max} - b_{max})$, where m_{max} denotes the largest length of this field among malicious inputs.

Alternatively, the maximum length could be too short, leading to false positives. To reduce their likelihood, our approach compares the maximum length computed as described above with b_{max}. If $b_{max} > l$ then `MAXLEN` is set to the geometric mean of b_{max} and m_{min}, where m_{min} denotes the minimum length of this field among all malicious inputs.

Clearly, the signatures described above have a simple structure, and hence may not be able to address complex vulnerabilities. In particular, vulnerabilities whose exploitation relies on the relationship between multiple message fields are not handled. For instance, there is a class of buffer overflows where a vulnerable program reads the value of a message field, and uses it to allocate a buffer, and copies another message field into this buffer [20]. More sophisticated machine learning techniques could potentially address such vulnerabilities. However, our focus in this paper has been on post-crash analysis techniques that leverage modern memory corruption defenses, and showing the feasibility of

Vulnerability description	Field name	Attack target	Gap size (Bytes)	MAX benign length	Attack length	Signature length
16-byte stack BO, no /GS	*method*	RA	0	9	1.1K	48
16-byte stack BO with /GS	*method*	RA	0	9	1.1K	48
16 byte stack BO, 32-byte local array	*method*	RA	32	9	1.1K	16
32-byte stack BO	*source*	SEH	0	15	1.1K	36
32-byte stack BO, 4-byte local integer	*source*	SEH	4	15	1.1K	32
64-byte Heap BO Freelist[00]	*data*	UEF	0	60	1.1K	64
64-byte Heap BO Freelist[1-127]	*data*	UEF	0	60	1.1K	64
260-byte Heap BO Lookaside list	*data*	SEH	0	256	1.1K	260
128-byte Heap BO triggering coalesce	*data*	RA	0	112	1.1K	128

Figure 2. Generation of Input Filters for Synthetic Application MEVS.

using these techniques for light-weight signature generation.

IV. System Implementation

A. User Environment

We assume our system will run on a typical Windows platform, Windows XP or Vista. Our focus is on applications such as a web server, FTP server and other network tools that are popular targets for remote attacks. Our system does not assume the availability of source code or debug symbols.

B. System infrastructure

We use Detours [8] from Microsoft Research to implement interceptors for input capture and filtering (currently, network inputs). Detours is a library for intercepting arbitrary Win32 binary functions on x86 machines. Detours are inserted at execution time, with the code of the target function modified in memory and the original target's functionality still available as a general subroutine. We used Detours to intercept network socket APIs and capture network input to a program. The input is passed to LSA through a named pipe. We also intercepted malloc to keep track of memory blocks in the heap.

For LSA, we used the Windows Debug Engine (WDE) [12] to implement our analysis techniques. This choice was motivated by the facts that the same debug engine is used for both Windows user and kernel level debugging, and it can support both live debugging and post-mortem debugging using the same API. WDE also supports remote debugging so that post-crash analysis can be done at a remote site instead of local host. WDE is used by popular Microsoft debuggers such as Windbg, Visual Studio and other command line debuggers like KD, CDB and NTSD.

V. Evaluation

Our evaluation made use of synthetic as well as real-world vulnerabilities. The former were incorporated in a synthetic vulnerable server MEVS (Memory Error Vulnerability Server), which provides a "string processing" service. A request to the server has three fields: a *method,* which specifies the operation to be performed by the server, and is one of {"ECHO", "REVERSE", "UPPERCASE", "LOWERCASE"}; a *source,* which is a machine name or an IP address; and *data*, which provides the operand for the operation specified by *method.* MEVS design is extensible, allowing new vulnerabilities to be introduced easily. It is designed in a way that simplifies the use of Metasploit to attack these vulnerabilities.

The use of synthetic server enabled us to experiment with many different vulnerabilities and exploit variations, including: use of working payload vs random data, use of text string vs binary data, enabling of and disabling of /GS protection, overwrite of RA (return address) vs SEH vs Function Pointer, exploits that lead to gaps and those that lead to no gaps, etc. For heap based buffer overflow, we experimented with neighboring block in four different locations: Freelist [00], Freelist [1 - 127], Look-aside list and overflow to trigger coalescing of heap blocks.

Figure 2 summarizes our results for MEVS. Signatures were generated for all exploits of the synthetic vulnerabilities. For stack buffer overflows, signature precision is improved when a gap is identified. For instances, in the case of 16-byte stack buffer overflow, after identifying the gap, our system generates a signature of length 16, which corresponds to the buffer boundary. This length is more precise than the 48-byte length that was generated in the absence of a gap. Similarly, a gap caused by a local integer increased the signature precision too. For heap based buffer overflow, our CTI and input correlation steps are very accurate in determining the maximum length from the length of the heap block involved in the overflow.

Figure 3 summarizes our results for real-world vulnerabilities on popular applications such as IIS and FTP. Each of the vulnerabilities shown in this figure are described in more detail below.

The first vulnerability we examined was a stack buffer overflow in the w3who ISAPI extension DLL (CVE-2004-1134). The w3who.dll library is a utility designed to provide auditing of server configuration remotely through a Web

Target application	Vulnerability identification	Field name	Gap size (bytes)	MAX benign length	Attack length	Signature length
IIS5	CVE-2004-1134	QueryString	0	20	8.0K	248
FreeFTPd 1.08	OSVDB-20909	user	0	14	1.8K	1002
Quick TFTP Pro 2.1	BugTraq-28459	mode	0	4	1.3K	1004
Steamcast 0.9.75	CVE-2008 0550	User-Agent	0	16	1.1K	1005
POP Peeper 3.4.0.0	BugTraq-34192	From	8	32	1.1K	328
TalkativeIRC 0.4.4.16	BugTraq 34141	PRIVMSG	30	56	1.0K	232

Figure 3. Generation of Input Filters for Real World Applications.

browser. When a long parameter is passed in as http query string, a stack buffer overflow results. Our working exploit targets Windows 2000 and Windows XP (SP2), and uses a stack overflow to overwrite the return address and/or SEH.

Real world vulnerability OSVDB-20909, is a stack buffer overflow in a freeware FreeFTPd version 1.0.8, at its multi-protocol file transfer service. The vulnerability is related to handling the field of "user". The exploit overwrites a stack buffer in freeFTPd service, and involves return address and SEH overwrite as well.

Quick Tftp Server Pro version 2.1 has a buffer-overflow vulnerability. Because the application fails to properly bounds-check user-supplied data for the "mode" field, an attack input can overflow a stack buffer and overwrite the return address or SEH.

Steamcast 0.9.75 and prior versions have a stack buffer overflow vulnerability (CVE-2008 0550) that enables attacks using the http User-Agent field to overwrite return address.

POP Peeper 3.4.0.0 is vulnerable in its "From" field. This vulnerability can lead to stack buffer overflow that can be used to overwrite the return address/SEH.

TalkativeIRC 0.4.4.16 has a vulnerability (BugTraq Id 34141) that allows the return address and SEH handler to be overwritten by supplying an excessively long value for the "PRIVMSG" field.

As mentioned in Section II, the advent of memory exploit defenses on modern Windows systems have made heap overflows difficult, thus prompting attackers to focus on stack overflows. This explains why our real-world exploits were all stack overflows.

In all 6 real world cases, our experiments identified the vulnerable buffer and generated blocking signatures. In 2 of these 6 cases, a gap is identified, and it makes the signature more precise.

A. False Positives

It is important for signatures to have very low (if not zero) false positive rate. Since our signature generation mechanism was able to infer the buffer bound from the location of the corruption target and/or the gap, we believe that they will not lead to false positives. Our experimental results support this claim as well: as shown in Figure 3, there is a large difference between maximum benign size and minimum attack size of vulnerable fields. In addition, we performed a false positive analysis by running the Apache and FTP servers with signatures turned on, and did not experience any false positives.

B. False Negatives

It is desirable to avoid false negatives, but we consider them to be less serious than false positives, provided that the signature can be refined, and false positives avoided after a small number of refinement steps. We already described such a refinement technique that converges in at most $\log m_{max}$ steps, where m_{max} is the largest size of vulnerable field across all benign inputs.

Experimentally, our evaluation shows a large difference between m_{max} and the signature length for 4 out of 6 real-world exploits. For the remaining two, the difference, though not as large, still remains significant.

False negatives may also occur because the underlying vulnerability is too complex to be addressed using the simple signature format that we use — in this case, it is likely that signature generation will fail, a limitation noted earlier.

C. Performance

Runtime overhead. The main runtime overhead is to capture and filter inputs at "detoured" input functions, as well as malloc-related functions. Since the intercepted inputs are maintained in memory, and filtering is based on length, malloc related logging only involves keeping start and size information in a table, the runtime overhead is relatively small. We measured this overhead for the freeFTPd server version 1.0.8. In particular, we measured the download time for 6 files of sizes are 1.7KB, 17KB, 110KB, 7.9MB, 23MB and 29MB. For base line test, the original application ran without any protection (no interception or any protection component attached). We fetched the same set of files from the FTP server, repeated 5 times, cleaning up cache between runs, and then calculating the average CPU time (sum of kernel cpu time and user cpu time) used. The baseline CPU time was 212.2ms. We repeated the same action with protections turned on, and the CPU time was 220ms. The overall overhead was thus about 4%.

Signature generation time. Security analysis is performed only at crash time, and hence its performance overhead is not a serious concern. For this reason, we have not done significant performance optimization on signature generation as yet. Nevertheless, we target the overhead to be low enough — roughly in the sub-second range.

Signature generation is fastest in the case of /GS and /SafeSEH protections. In these cases, the CTI phase searches only a few locations, and correspondingly, the number of calls to approximate string matching are also small. As a result, signature generation is very fast, taking about a millisecond in our experiments.

If the exception is caused by ASR or NX and it involves a stack overflow, then the search takes longer, as the entire stack may need to be scanned. In our experiments, this case typically takes about 100 milliseconds.

If the exception involves heap corruption, then a much larger number of locations are scanned by our current implementation. Due to the rarity of heap overflows, this case has not been optimized at all, and hence takes about 800ms.

The measurements for both runtime overhead and signature generation time were performed on a VirtualPC 2007 with a guest Windows XP SP2 guest OS and 256MB RAM running on a DELL M4300 with 2GB RAM and Windows XP SP2 host OS.

VI. Related Work

Windows Error Report is the first step for application crash response provided by Microsoft. Abouchaev et al [1] present an excellent bug fix guideline for Microsoft internal developers regarding crash and exploitability. According to this document, different types of exceptions have different potentials for exploitability. Based on this work, an exploitable crash analyzer built as a Windows debugger extension is developed [17]. The tool provides crash categorization and exploitability assessment. It is targeted at developers and is mainly used in Microsoft internal fuzz testing to identify vulnerabilities. The tool assumes that the information in the faulting instruction is controlled by an attacker, but does not attempt to relate crash back to input.

Slowinska and Bos [20] identified the “gap” problem and proposed adding a new dimension — timestamp — to taint analysis to solve the problem. Though their approach may be more general, it is not applicable for online usage due to high overhead of taint-tracking.

Researchers have used program structural constraints for automated security debugging in Linux debugging environment [9]. It may be possible to apply program structural constraints to enhance our initial corruption target identification if these constraints can be identified reliably without access to source code.

There have been a number of research efforts on generating blocking signatures for network worms. Much of this work focused on “content-based signatures,” where the signature captured characteristics of the attack payload. The problem with such signatures is that attackers can easily evade these signatures since they have full control over the payload. In contrast, the underlying vulnerabilities are dependent on the implementation details of a victim application. Attackers have no control over these vulnerabilities, and have very little choice in terms of the methods for exploiting them. Thus, signature generation techniques that focus on vulnerabilities can be much more resistant to evasion.

Brumley et al [2] developed an approach for generating vulnerability oriented signatures based on symbolic execution techniques. While the techniques work well to capture features that are closely related to the program paths exercised by an exploit, generalization of the technique to the case where other vulnerable paths are considered remains to be a challenge. As a result, for real-world applications, the signatures generated by this technique are not general enough.

Unlike [2], which relies on a whitebox approach for signature generation, COVERS [10] uses a blackbox technique that exploits the features of buffer overflow exploits to generate vulnerability oriented signatures for them. Another difference between [2] and [10] is that the latter aims to generate signatures very quickly so that they could be deployed immediately, thus preserving server availability. The work presented in this paper is similar to COVERS in its use of post-crash memory analysis, but differs from it in several significant ways. First, COVERS uses exact matching only, so it can fail to generate signatures when a “gap” is present. Second, the approach presented in this paper is able to reason about likely buffer lengths, and generate signatures based on these. In contrast, COVERS signatures are based only on comparing the sizes of benign and attack inputs, and are hence less accurate. Finally, COVERS is focused on Linux, whereas the work presented in this paper targets Windows.

Several researchers have focused on developing techniques to generalize signatures. PacketVaccine [21] uses a randomization-based approach to generate variants of an exploit and uses these variants to ensure that a generalized signature is generated. ShieldGen [7] develops a more systematic approach for generating attack variants by exploiting a precise specification of underlying network protocols. Both these techniques are complementary to our approach, and represent a different tradeoff in terms of speed of signature generation versus generality.

VII. Conclusion

We presented a new signature generation technique for Windows-based applications that is light-weight enough to be deployed on production systems. The online nature of our system enables it to provide protection from a wide range of memory corruption exploits, including those involving

unknown vulnerabilities, or known vulnerabilities but unknown exploits. In contrast, most previous techniques need to be run in sandboxed environments, and need working exploits to generate signatures. Our technique uses taint-inference [14] rather than heavy-weight taint analysis. It uses an innovative post-crash analysis of victim process memory and approximate substring matching to reason about likely lengths of vulnerable buffers, and uses this information to improve signature accuracy. Moreover, our technique is able to handle "gaps" that are commonly associated with Windows exploits, and posed a challenge to many previous signature generation techniques. Our experimental results are promising, and demonstrate that the approach can generate accurate signatures for many real-world servers.

VIII. Acknowledgements

We would like to thank Mark Cornwell for his contributions in testing and evaluation.

References

[1] A. Abouchaev, D. Hasse, S. Lambert, and G. Wroblewski. CRASH COURSE-Analyze Crashes To Find Security Vulnerabilities In Your Apps. *MSDN Magazine-Louisville*, pages 60–69, 2007.

[2] 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*, pages 2–16. IEEE Computer Society Washington, DC, USA, 2006.

[3] Shuo Chen, Jun Xu, and Emre Can Sezer. Non-control-hijacking attacks are realistic threats. In *USENIX Security Symposium*, 2005.

[4] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: Securing software by blocking bad input. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 117–130. ACM New York, NY, USA, 2007.

[5] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. *ACM SIGOPS Operating Systems Review*, 39(5):133–147, 2005.

[6] Crispin Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, pages 63–78, San Antonio, Texas, January 1998.

[7] W. Cui, M. Peinado, H.J. Wang, and M.E. Locasto. Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *IEEE Symposium on Security and Privacy, 2007. SP'07*, pages 252–266, 2007.

[8] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, pages 135–143, 1999.

[9] C. Kil, E.C. Sezer, P. Ning, and X. Zhang. Automated Security Debugging Using Program Structural Constraints. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 453–462, 2007.

[10] 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*, pages 213–222. ACM New York, NY, USA, 2005.

[11] D. Litchfield. Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server. *NGSSoftware Ltd, http://www.nextgenss.com*, 2003.

[12] Microsoft. Debugger Engine and Extension APIs. *http://msdn.microsoft.com/en-us/library/cc267863.aspx*.

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

[14] R. Sekar. An Efficient Black-box Technique for Defeating Web Application Attacks. *NDSS*, 2009.

[15] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM New York, NY, USA, 2007.

[16] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *ACM Conference on Computer and Communication Security (CCS)*, pages 298–307, Washington, DC, October 2004.

[17] J. Shirk and D. Weinstein. Automated Real-time and Post Mortem Security Crash Analysis and Categorization. In *CanSecWest*, 2009.

[18] skape. Reducing the Effective Entropy of GS Cookies. *http://www.uninformed.org/?v=all&a=32&t=txt*, 2007.

[19] Skywing skape. Bypassing Windows Hardware-enforced DEP. *http://www.uninformed.org/?v=2&a=4&t=txt*, 2005.

[20] A. Slowinska and H. Bos. The age of data: pinpointing guilty bytes in polymorphic buffer overflows on heap or stack. In *23rd Annual Computer Security Applications Conference (ACSAC07)*.

[21] X.F. 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 conference on Computer and communications security*, pages 37–46. ACM New York, NY, USA, 2006.