# English Shellcode

Joshua Mason, Sam Small
Johns Hopkins University
Baltimore, MD
{josh, sam}@cs.jhu.edu

Fabian Monrose
University of North Carolina
Chapel Hill, NC
fabian@cs.unc.edu

Greg MacManus
iSIGHT Partners
Washington, DC
gmacmanus.edu@gmail.com

## ABSTRACT

History indicates that the security community commonly takes a divide-and-conquer approach to battling malware threats: identify the essential and inalienable components of an attack, then develop detection and prevention techniques that directly target one or more of the essential components. This abstraction is evident in much of the literature for buffer overflow attacks including, for instance, stack protection and `NOP` sled detection. It comes as no surprise then that we approach shellcode detection and prevention in a similar fashion. However, the common belief that components of polymorphic shellcode (e.g., the decoder) cannot reliably be hidden suggests a more implicit and broader assumption that continues to drive contemporary research: namely, that valid and complete representations of shellcode are fundamentally different in structure than benign payloads. While the first tenet of this assumption is philosophically undeniable (i.e., a string of bytes is either shellcode or it is not), truth of the latter claim is less obvious if there exist encoding techniques capable of producing shellcode with features nearly indistinguishable from non-executable content. In this paper, we challenge the assumption that shellcode must conform to superficial and discernible representations. Specifically, we demonstrate a technique for automatically producing *English Shellcode*, transforming arbitrary shellcode into a representation that is superficially similar to English prose. The shellcode is completely self-contained—i.e., it does not require an external loader and executes as valid `IA32` code—and can typically be generated in under an hour on commodity hardware. Our primary objective in this paper is to promote discussion and stimulate new ideas for thinking ahead about preventive measures for tackling evolutions in code-injection attacks.

## Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Invasive software


*CCS'09,* November 9–13, 2009, Chicago, Illinois, USA.


## General Terms

Security, Experimentation

## Keywords

Shellcode, Natural Language, Network Emulation

## 1. INTRODUCTION

Code-injection attacks are perhaps one of the most common attacks on modern computer systems. These attacks are used to deliver and run arbitrary code on victims' machines, often enabling unauthorized access and control of system resources, applications, and data. Typically, the vulnerabilities being exploited arise due to some level of neglect on the part of system and application developers to properly define and reject invalid program input. Indeed, the canonical consequences of such neglect, which include buffer and heap overflow attacks, format string attacks, and (more recently) heap spray attacks, categorically demonstrate some of the most popular code-injection techniques.

Generally speaking, an attacker's first objective in a code-injection attack is to gain control of a machine's program counter. The program counter is a special purpose machine register that identifies the next instruction scheduled for execution. By gaining control of the program counter, an attacker is able to redirect program execution and disrupt the intended behavior of the program. With the ability to manipulate the program counter, attackers sometimes redirect a victim's machine to execute (already present) application or system code in a manner beneficial to an attacker's intent. For instance, *return-to-libc* attacks provide a well-documented example of this kind of manipulation. In a code-injection attack, however, attackers redirect the program counter to execute code delivered by the attackers themselves. Depending on the details of the particular vulnerability that an attacker is targeting, injected code can take several forms including source code for an interpreted scripting-language engine, intermediate byte-code, or natively-executable machine code.

Despite differences in the style and implementation of different exploits, e.g., buffer overflow versus format string attacks, all code-injection attacks share a common component: the injected code. This payload typically provides attackers with arbitrary control over a vulnerable machine. Frequently (though not always), attackers deliver a payload that simply launches a command shell. It is for this reason that many in the hacking community generically refer to the payload portion of a code-injection attack as *shellcode*.

Among those less familiar with code-injection attacks, there is sometimes a subtle misconception that shellcode is necessarily delivered *in tandem* with whichever message ultimately exploits a vulnerability and grants an attacker control of the program counter. While this assumption typically holds in more traditional buffer overflow vulnerabilities, modern attacks demonstrate that attackers have developed numerous techniques to covertly deliver (and ultimately store into memory) shellcode separately from and prior to triggering the exploit. For instance, if an attacker can manipulate memory at a known heap address, they may store their shellcode there, using its address later when overwriting a return address on the stack [10]. We draw attention to this distinction because our use of the term *shellcode* in this paper specifically denotes the injected code irrespective of individual attacks or vulnerabilities.

Typically, shellcode takes the form of directly executable machine code, and consequently, several defensive measures that attempt to detect its presence, or prevent its execution altogether, have been proposed. Indeed, automated inspection of user input, system memory, or network traffic for content that appears statistically or superficially executable are now common (e.g., [23, 1, 16, 15, 27]). However, as expected, a number of techniques have been developed that circumvent these protective measures, or make their job far more difficult (e.g., polymorphism [5, 7]).

Recently, it has been suggested that even polymorphic shellcode is constrained by an essential component: the *decoder*. The argument is that the decoder is a necessary and executable companion to encoded shellcode, enabling the encoded portion of the payload to undergo an inverse transformation to its original and executable form. Since the decoder must be natively executable, the prevailing thought is that we can detect its presence assuming that this portion of the payload will bear some identifiable features not common to valid or non-executable data. It is this assumption—that shellcode is fundamentally different in structure than non-executable payload data—that continues to drive some avenues of contemporary research (e.g., [27, 16, 15, 26]).

By challenging the assumption that shellcode must conform to superficial and discernible representations, we question whether protective measures designed to assume otherwise are likely to succeed. Specifically, we demonstrate a technique for automatically producing *English Shellcode* — that is, transforming arbitrary shellcode into a representation that is statistically similar to English prose. By augmenting corpora-based natural-language generation with additional constraints uniquely dictated by each instance of shellcode, we generate encodings complete with decoder that remain statistically faithful to the corpus and yield identical execution as the original shellcode. While we in no way claim that instantiations of this encoding are irrefutably indistinguishable from authentic English prose—indeed, as shown later, it is clear they are not—the expected burden associated with reliably detecting English-encoded shellcode variants in juxtaposition to genuine payloads at line speed raises concerns about current preventative approaches.

Similar to the goal of Song et al. [20], our objective in this paper is to promote discussion and stimulate new ideas for thinking about how to tackle evolutions in code-injection attacks. Although most of the attacks observed today have used relatively naïve shellcode engines [17, 26], exploit code will likely continue to evade intrusion detection and prevention systems because malcode developers do not follow the "rules". As this cat and mouse game plays on, it is clear that the attackers will adapt. So should we, especially as it pertains to exploring new directions for preventitive measures against code-injection attacks.

## 2. ON THE ARMS RACE

In this paper, we focus on natively-executable shellcode for x86 processors. In this case, machine code and shellcode are fundamentally identical; they both adhere to the same binary representation directly executable by the processor.

Shellcode developers are often faced with constraints that limit the range of byte-values accepted by a vulnerable application. For instance, many applications restrict input to certain character-sets (e.g., printable, alphanumeric, `MIME`), or filter input with common library routines like `isalnum` and `strspn`. The difficulty in overcoming these restrictions and bypassing input filters depends on the range of acceptable input. Of course, these restrictions can be bypassed by writing shellcode that does not contain restricted byte-values (e.g., null-bytes). Although such restrictions often limit the set of operations available for use in an attack, attackers have derived encodings to convert unconstrained shellcode honoring these restrictions by building equivalency operations from reduced instruction sets (e.g., [25, 11]).

Of special note are the alphanumeric encoding engines [18] present in Metasploit (see `www.metasploit.com`). These engines convert arbitrary payloads to representations composed only of letters and numerical digits. These encodings are significant for two reasons. First, alphanumeric shellcode can be stored in atypical and otherwise unsuspected contexts such as syntactically valid file and directory names or user passwords [18]. Second, the alphanumeric character set is significantly smaller than the set of characters available in Unicode and `UTF-8` encodings. This means that the set of instructions available for composing alphanumeric shellcode is relatively small. To cope with these restrictions, patching or self-modification is often used. Since alphanumeric engines produce encodings automatically, a decoder is required. The challenge then is to develop an encoding scheme and decoder that use only alphanumeric characters (and hence, a restricted instruction set), yet are together capable of encoding arbitrary payloads. The top three rows in Figure 1 show examples using the Metasploit framework.

| ENCODING | HEX | ASCII |
|---|---|---|
| *None* | 31DB5343536A ... | 1#SCSj#jfX######CRfh\fS##jfXPQV####... |
| *PexAlphaNum* | 515A56545836 ... | QZVTX630VX4A0B6HH0B30BCVX2BDBH4A2AD... |
| *Alpha2* | 374949515A6A ... | 7IIQZjJX0B1PABkBAZB2BA2AA0AAX8BBPux... |
| *English* | 546865726520 ... | There is a major center of economic... |

*Figure 1:* Example encodings of a Linux `IA32` Bind Shell. The *PexAlphaNum* and *Alpha2* encodings were generated using the Metasploit Framework. A hash symbol in the last column represents a character that is either unprintable or from the extended ASCII character set.

We note that much of the literature describing code injection attacks (and prevention) assumes a standard attack template consisting of the basic components found traditionally in buffer-overflow attacks: a `NOP` sled, shellcode, and one

or more pointers to the shellcode [1, 12, 23, 27]. Not surprisingly, the natural reaction has been to develop techniques that detect such structure or behavior [20, 23, 16, 15, 27, 14]. While emulation and static analysis have been successful in identifying some of the failings of advanced shellcode, in the limit, the overhead will likely make doing so improbable. Moreover, attacks are not constrained to this layout and so attempts at merely detecting this structure can be problematic; infact, identifying each component has its own unique set of challenges [1, 13], and it has been suggested that malicious polymorphic behavior cannot be modeled effectively [20]. In support of that argument, we provide a concrete instantiation that shows that the decoder can share the same properties as benign data.

# 3. RELATED WORK

Defensive approaches against code-injection attacks tend to fall into three broad categories. The first centers around tools and techniques to both limit the spoils of exploitation and to prevent developers from writing vulnerable code. Examples of such approaches include automatic bounds protection for buffers [4] and static checking of format strings at compile-time, utilizing "safe" versions of system libraries, and address-space layout randomization [19], etc. While these techniques reduce the attack surface for code-injection attacks, no combination of such techniques seems to systematically eliminate the threat of code-injection [6, 21].

In light of persistent vulnerabilities, the second category of countermeasures focuses on preventing the execution of injected code. In this realm, researchers have demonstrated some success using methods that randomize the instruction-set [22] or render portions of the stack non-executable. Although these approaches can be effective, instruction-set randomization is considered too inefficient for some workloads. Additionally, recent work by Buchanan *et al.* demonstrates that without better support for constraining program behavior, execution-redirection attacks are still possible [3].

The third category for code-injection defense consists of content-based input-validation techniques. These approaches are either host or network-based and are typically used as components in intrusion detection systems. User-input or network traffic is considered suspicious when it appears executable or anomalous as determined by heuristic, signature, or simulation.

In this area, Toth and Kruegel detect some buffer overflow exploits by interpreting network payloads as executable code and analyzing their execution structure [23]. They divide machine instructions into two categories separated by those that modify the program counter, i.e., *jump* instructions, and others that do not. Their experiments show that, under some circumstances, it is possible to identify payloads with executable code by evaluating the maximum length of instruction sequences that fall between jump instructions, and find that payloads with lower maximum execution lengths are typically benign. However, their evaluation does not include an analysis of polymorphic code, and Kolesnikov et al. show that polymorphic blending attacks evade this detection approach [9].

Several approaches have been suggested for identifying self-decrypting shellcode using emulation [15, 27, 2] or dynamic taint analysis [26]. However, these detection methods are based on a number of assumptions that do not necessarily need to be so. For instance, they detect decryption routines in polymorphic code by scanning network traffic for `GetPC` code. Essentially, this includes any instructions that provide an attacker with the value of the instruction pointer (e.g., using the `fstenv` instruction). They reason that some form of `GetPC` code is necessary for determining the location of an exploit's encrypted payload. However, many exploits do not follow this convention and attackers can often determine the location of their payload by simply understanding how a particular exploit affects machine state or by manipulating it themselves as part of the attack. Furthermore, these emulation techniques are incomplete because they cannot accurately reproduce the behavior of execution candidates without register and flag information – information that is unavailable to network-based intrusion detection systems.

Polychronakis et al. address some of these limitations by examining shellcode without `GetPC` code, coined non-self-contained polymorphic shellcode [16]. By developing a number of behavioral heuristics, they were able to identify polymorphic shellcode by emulating execution from data in numerous network traces. While their approach significantly improves upon previous detection methods, the contents of memory and registers are still unknown, making accurate emulation a challenge. For instance, an attacker may know the value of registers (e.g., `EFLAGS`) or memory accessible from the vulnerable process prior to shellcode execution. This means that an attacker can use conditional jumps or other operations to obfuscate the execution path of the shellcode. In particular, self-modifying shellcode that is laced with conditional operations raises challenges for emulation-based techniques as they must execute all possible execution paths. While path enumeration can be tractable in certain shellcode encodings where conditional statements are rare, the English letters "p" through "z" are all conditional jumps. Therefore, when it comes to English, shellcode designed in tandem with the exploit makes current emulation particularly difficult.

Lastly, Song et al. examine popular polymorphic shellcode engines to assess their strengths and weaknesses [20]. Our work supports their observations in that while today's polymorphic engines do generate observable artifacts, these artifacts are not intrinsically symptomatic of polymorphic code. However, while they advise that modeling acceptable content or behavior may lead to a better long-term solution for preventing shellcode delivery, we argue that even modeling acceptable content will be rife with its own set of challenges, as exemplified by English shellcode. Specifically, by generating malicious code that draws from a language model built using *only* benign content, statistical measures of intent become less accurate and the signal-to-noise ratio between malicious code and valid network data declines.

# 4. TOWARDS ENGLISH SHELLCODE

Shellcode, like other compiled code, is simply an ordered list of machine instructions. At the lowest level of representation, each instruction is stored as a series of bytes signifying a pattern of signals that instruct the CPU to manipulate data as desired. Like machine instructions, non-executable data is represented in byte form. Coincidentally, some character strings from the ASCII character and native machine instructions have identical byte representations. Moreover, it is even possible to find examples of this phenomenon that parse as grammatically correct English sentences. For instance, ASCII representation of the phrase "Shake Shake

Shake!" is byte-equivalent to the following sequence of Intel instructions: `push %ebx; push "ake "; push %ebx; push "ake "; push %ebx; push "ake!"`.

However, it is unlikely that one could construct meaningful code by simply concatenating English phrases that exhibit this property. Abiding by the rules of English grammar simply excludes the presence of many instructions and significantly limits the availability and placement of others. For example, `add`, `mov`, and `call` instructions cannot be constructed using this method. Therefore, while it may be possible to construct some instances of shellcode with coherent objectives in this manner, the versatility of this technique is severely restricted by its limitations. Rather than find these instances, our goal is instead to develop an automated approach for transforming *arbitrary* shellcode into an English representation.

## 4.1 High-level Overview

What follows is a brief description of the method we have developed for encoding arbitrary shellcode as English text. This *English shellcode* is completely self-contained, i.e., it does not require an external loader, and executes as valid `IA32` code. The steps depicted in Figure 3 complement the brief overview of our approach presented below. One can envision a typical usage scenario (see Figure 4) where the English shellcode (composed of a natively executable decoder and an encoded payload containing arbitrary shellcode) is first generated offline. Once the English shellcode is delivered to a vulnerable machine and its vulnerability is triggered, execution is redirected to the English shellcode, initiating the decoding process and launching the target shellcode contained in the payload.

First, a list of English-compatible instructions were compiled and categorized loosely by behavior, i.e., whether an instruction performs a jump, executes a boolean operation, or manipulates the stack. Some excerpts from the list are shown in Figure 2. Using this list and its categorization to guide development, a decoder was written that is capable of encoding generic payloads using only instructions from our list. This intermediate result is similar in spirit to the alphanumeric decoders, however, our decoder is further constrained by a guiding principle to avoid certain character patterns that might later make finding an English equivalent more difficult, e.g., the string of mostly capital letters that compose the *PexAlphaNum* and *Alpha2* decoders depicted in Figure 1 would likely result in poor English shellcode.

The basic idea then is to find a strings of English words that mimic the execution behavior of our custom decoder. To achieve this goal, we use a smoothed $n$-gram language model. That model is trained using a large set of English text, and is used to deduce the probability of word sequences. As language generation proceeds, each instruction in the decoder is assigned a numerical value. Intuitively, as we select candidate strings from the language model, each is executed under supervision. We use the numerical values to indicate the strength of each candidate. If a candidate string produces the same net effect of the first instruction of our decoder when executed, we say that its score is one. If a candidate string produces the net effect of the first two instructions, its score is two (and so on). At each stage, high-scoring candidates are kept (and used in driving the language model forward) and low-scoring candidates (or those that crash the simulator) are discarded. Ultimately, we traverse the language model using a beam search to find strings of words that score the highest possible value and operate in an identical manner as the decoder developed by hand.

**STORAGE**

| ASCII | HEX | ASSEMBLY |
|---|---|---|
| " ca" | 20 63 61 | and 61(%ebx), %ah |
| " An" | 20 41 6E | and 6E(%ecx), %al |
| " jo" | 20 6A 6F | and 6F(%edx), %ch |

**JUMPS**

| ASCII | HEX | ASSEMBLY |
|---|---|---|
| p. | 70 2E | jo short $30 |
| q. | 71 2E | jno short $30 |
| r. | 72 2E | jb short $30 |
| s. | 73 2E | jnb short $30 |
| t. | 74 2E | je short $30 |
| u. | 75 2E | jnz short $30 |
| v. | 76 2E | jbe short $30 |
| w. | 77 2E | ja short $30 |
| x. | 78 2E | js short $30 |
| y. | 79 2E | jns short $30 |
| z. | 7A 2E | jpe short $30 |

**STACK MANIPULATION**

| ASCII | HEX | ASSEMBLY |
|---|---|---|
| A | 41 | inc %eax |
| B | 42 | inc %edx |
| C | 43 | inc %ebx |
| D | 44 | inc %esp |
| E | 45 | inc %ebp |
| F | 46 | inc %esi |
| G | 47 | inc %edi |
| H | 48 | dec %eax |
| I | 49 | dec %ecx |
| J | 4A | dec %edx |
| K | 4B | dec %ebx |
| L | 4C | dec %esp |
| M | 4D | dec %ebp |
| N | 4E | dec %esi |
| O | 4F | dec %edi |
| P | 50 | push %eax |
| Q | 51 | push %ecx |
| R | 52 | push %edx |
| S | 53 | push %ebx |
| T | 54 | push %esp |
| U | 55 | push %ebp |
| V | 56 | push %esi |
| W | 57 | push %edi |
| X | 58 | pop %eax |
| Y | 59 | pop %ecx |
| Z | 5A | pop %edx |
| a | 61 | popa |

*Figure 2:* Byte-values that have valid interpretations as English ASCII characters and Intel assembly instructions.

Finally, to encode the original payload, we continue to sample strings from our language model all the while generating prose that is functionally equivalent to the target shellcode when executed.

## Our Approach

Recall that unlike other attack components, the decoder must reside in memory as executable code. This exposure can make identifying the decoder a useful step in facilitating detection and prevention (e.g., by determining if a portion of the payload "looks" executable). Thus, from an attacker's perspective, masking the decoder may reduce the likelihood of detection and help to facilitate clandestine attacks.

Designing a decoder under unnatural constraints can be very challenging, and this difficulty is not unique to English shellcode. Self-modification is often used to address this problem whereby permissible code modifies portions of the payload such that non-compliant instructions are patched in at runtime, thereby passing any input filters. These additional instructions provide an attacker with more versatility and may make an otherwise impotent attack quite powerful.

Self-modification is particularly useful for overcoming some of the challenges unique to English shellcode. Among the English-compatible instructions, for example, there is no native support for loops or addition. Issues like these are relevant because decoding a payload without certain instructions, while possible, can quickly become impractical. For instance, a decoder without a looping mechanism must be proportional in length to the length of its encoded payload, possibly exposing its existence by nature of its size and form on the wire.

## 4.2 The decoder

We are able to avoid these problems by building a self-modifying decoder that has the form: `initialization, decoder, encoded payload`. Intuitively, the first component builds an initial decoder in memory (through self-modification) which when *executed*, expands the working instruction set, providing the decoder with `IA32` operations beyond those

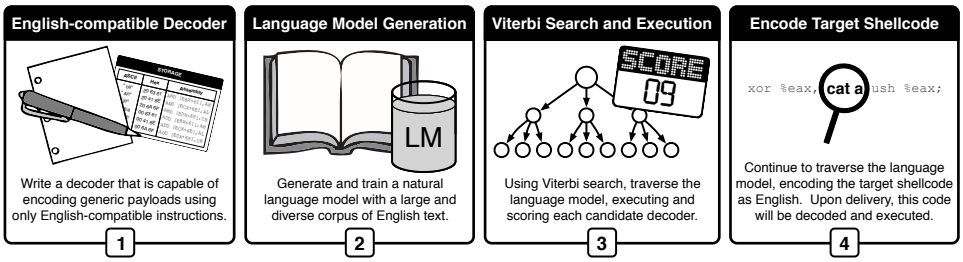


*Figure 3:* Our method of generating English shellcode consists of four distinct components: developing an English-compatible decoder, constructing a large $n$-gram language model, scoring candidate execution, and encoding arbitrary shellcode.

provide by English prose. The decoder then decodes the next segment (the encoded payload), again via self-modification.

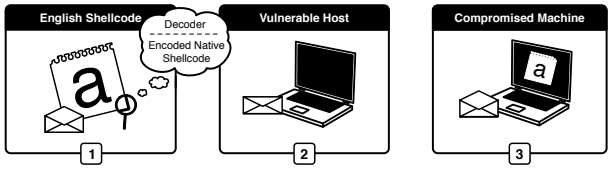


*Figure 4:* A typical usage scenario.

We build our decoder using a number of principles that help guide its design. First and foremost, the decoder must use only English-compatible instructions or the goal of creating English shellcode cannot be realized. Furthermore, we are particularly interested in English-compatible instructions that can be used, alone or in conjunction, to produce useful instructions (via self modification) that are not English-compatible. For example, our decoder uses multiple `and` instructions (which are English-compatible) to generate `add` instructions (which are not English-compatible). Taken together, it could be said that these first two goals also provide a foundation for the design of alphanumeric decoders. However, our third design principle, which is not shared by alphanumeric shellcode engines, is to favor instructions that have less-constrained ASCII equivalents. For instance, we will likely favor the instruction `push %eax` ("P") over `push %ecx` ("Q") when designing our decoder since the former is more common in English text. The same guiding principle is applied when choosing literal values.[1] It is important to note that even though we followed these principles in designing our decoder, they are not hard requirements and there are other capable approaches. What we provide here is a proof of concept to demonstrate our point.

*Initialization.* In its initialization phase, the decoder overwrites key machine registers and patches in machine instructions that are not English compatible. After successful exploitation of a software vulnerability, we assume that a pointer to the shellcode resides in one of the general purpose registers or other accessible memory. As pointed out by Polychronakis et al., this is common in non-self-contained shellcode [16]. In order to execute the target shellcode, a pointer to the encoded shellcode is needed. This pointer must address memory far beyond the first byte of the shellcode since one must first reserve space for the decoder.

[1]The term *literal* refers to a numerical operand in this context.

| STEP | INSTRUCTION | ASCII OPTIONS | STACK |
|---|---|---|---|
| 1 | `push ptr; push ptr;` | $x \in$ [P, W] | DE AD BA EF DE AD BA EF 6B 02 A4 |
| 2 | `inc %esp;` | D | - DE AD BA EF DE AD BA EF 6B 02 A4 |
| 3 | `pop reg; pop reg;` | $x \in$ {X, Y, Z, a} | - DE AD BA EF DE AD BA EF 6B 02 A4 |
| [4, $n$ - 2] | `inc reg; inc reg; ...` | $x \in$ [A, G] | NO STACK EFFECT |
| $n$ - 1 | `push reg;` | $x \in$ [P, W] | - DE AD BA EF DE AD BE EF 6B 02 A4 |
| $n$ | `dec %esp;` | L | DE AD BE EF DE AD BE EF 6B 02 A4 |

%esp

*Figure 5:* Leveraging the stack to increment the pointer to our payload. In this example, the pointer value is initially `0xDEADBAEF` and is increased by 1024 to `0xDEADBEEF`.

Since the register containing the address of the shellcode is known, we can copy its pointer and add an offset to reach the encoded payload. Using only English-compatible ASCII characters, the increment instruction `inc` is the most obvious candidate for increasing a register's value. However, this one-byte instruction will only increase the value of a register in increments of one, yielding no space for the decoder. Used this way, the `inc` instruction is insufficient.

However, a single `inc` instruction can be used to increase a register value in increments of 256 after manipulating the alignment of the stack. This process is depicted in Figure 5. For instance, we can first push the shellcode pointer onto the stack and shift the stack pointer `%esp` by one byte. Once shifted, the `pop` instruction places the three least-significant bytes of the shellcode pointer into a register where its value is increased using `inc` multiple times.[2] Afterwards, the value of this register is pushed back onto the stack and the stack is realigned. The top of the stack, which at first contained the shellcode pointer, now contains the same value increased by increments of 256. By popping this value into a register, we can use it to address the encoded payload.

*Unpacking the decoder.* To facilitate looping, instructions that are not English compatible (e.g., the `lods` and `add` instructions) are needed. However, to generate these, the shellcode can manipulate its contents to patch in the required instructions. For example, an `and` instruction can be used to create an `add` instruction. The opcode for `and` is equivalent to the ASCII space character (`0x20`), which

[2]Respectively, we `push` and `pop` the shellcode pointer twice in a row to avoid having an unpredictable byte (i.e., the byte marked "-" in step 2 from Figure 5) in the register we increment. This follows one of our more general principles discussed in Section 5: avoid operations that may set flags in an unpredictable manner.

is convenient because the space character is the most common character in English. The variant used in our proof-of-concept is three bytes in length and takes the form "`AND r/m8, r8`". Its first parameter, `r/m8`, addresses the bytes to be modified, while the second specifies one of the partial registers. The opcode for the `add` operations we create is `0x00`. This means that the partial register and the byte addressed by the `r/m8` operand must yield a value of zero when the `and` operation is executed. Thus, the partial registers used are chosen such that a zero byte is created at `r/m8`.

**Decoder Loop**

```
1:  top:
2:      sub $0x20,%al
3:      jnz decode
4:      inc %edi
5:
6:  decode:
7:      lods
8:      add %ah,74(%edi)
9:      jnz top
```

*Figure 6:* Example decoder loop. Blocks of English words in the payload are read by the decoder and transformed into arbitrary executable machine code.

After patching, the `add` instructions further help build the decoder by patching in the load string instruction, `lods`. The load string instruction is used to read bytes from the encoded payload. It loads into `%eax` four bytes from the address referenced by `%esi` (i.e., the encoded payload) and afterwards, immediately increments the address by four.

*Decoding.* The decoder loop, shown in Figure 6, reads blocks of English words from the encoded payload, decoding the target shellcode byte-by-byte. For space efficiency, the target shellcode overwrites the encoded payload as it is decoded. To facilitate this process, two pointers are initialized to point to the encoded payload. The first pointer, `%esi`, is implicitly used to identify the current read position via the `lods` instruction and the other pointer, an offset from `%edi`, marks the output position for the next decoded byte. As described subsequently, the second pointer is also used as an accumulator.

The decoder reads data from the encoded payload in contiguous four-byte blocks. The first two bytes of each block are ignored. The value of the third byte in each block is added to the value referenced by the output pointer, which initially points to the first character of the encoded shellcode. If the value of the fourth byte in a block is equivalent to the space character, accumulation ends and the output pointer is advanced. This process ultimately terminates when accumulation yields the null character (`0x00`). At its conclusion, the target shellcode is completely decoded in memory, located in the same position that the encoded payload originally resided.

The stop condition for both decoding individual bytes and the decoder loop itself are controlled by the conditional jumps shown at lines 3 and 9 of the assembly listing in Figure 6. Since the jump-if-not-zero instruction, `jnz`, is controlled by the zero-flag, the first jump is influenced by the outcome of the subtraction in the previous line (the zero-flag is set if the difference between its two operands, `0x20` and `%al`, is zero). These two instructions are easily supported by English-compatible characters: the subtraction instruction, `sub`, has the same byte representation as a comma before a space character (i.e., ", "), while `jnz` is the English character "u". Similarly, the second jump is influenced by the `add` operation that immediately precedes it and was patched in by the decoder itself.

The decoder presented in this section is composed of byte values that are particularly helpful in facilitating the creation of English shellcode. That said, there are many other ways to accomplish the same task using other series of instructions. If a detection method is developed for one decoder in particular, it can trivially be replaced with another that performs the same operations using different instructions.

*Initializing registers.* To ease the aforementioned exposition, we omitted discussion on how registers are initialized. At the same time that the address of the encoded shellcode is moved to a register (before the decoder loop executes), several other values are stored. Specifically, the `popa` instruction, whose opcode is equivalent to the character "a", allows us to set the contents of the registers used by the `and` instruction (to create an `add` instruction) and the `add` instruction (to create the `lods` instruction). The `popa` instruction pops 32 bytes from the stack into 8 registers. The registers' values are set by pushing the values on the stack before `popa` executes. The two `push` operations we use push either one or four bytes onto the stack and are equivalent to the characters "j" and "h", respectively. For example, the word, "johnboat", first pushes "`o`" and then "`nboa`" onto the stack.

# 5. AUTOMATIC GENERATION

Recall that the instructions used to implement the decoder are selected specifically because their byte-representations match those of characters used commonly in English-based ASCII strings. Taken as-is, the custom decoder will have common English characters, but will not have the appearance of English text.

Intuitively, there are three general types of instructions that give us the freedom to position the decoder instructions among English words and produce multiple variants of English shellcode from the same payload. The first type includes all English-compatible instructions that produce no net execution effect, i.e., `nop`. Second are operations that may in general affect machine state, but for our purposes will not interfere with the operation of the decoder (or the decoding process). The last type are those that in series may intermediately affect machine state in an undesirable manner, but taken in sum, have the same net effect on machine state as the other two types, i.e., no effect of consequence or no effect at all.

To generate an English-like decoder *automatically*, we use techniques that draw heavily from the natural language processing community, augmenting a statistical language generation algorithm with additional constraints. The language generation architecture is influenced by statistical information about the target language, i.e., English, by observing its use in various settings. We use a corpus comprised of just over fifteen thousand Wikipedia articles and roughly $27,000$ books from the Project Gutenberg (see `www.gutenberg.org`) to train a statistical model, termed a *language model*, which contains counts of words and word sequences observed during training.

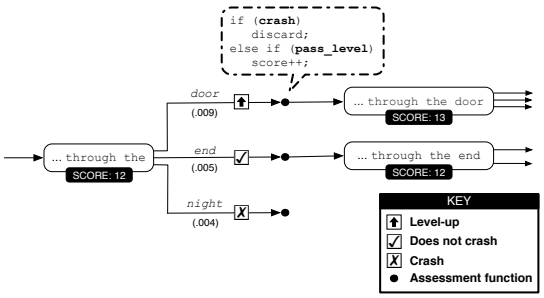


*Figure 7:* An English version of the decoder is found by sampling the language model for words that contribute to the achievement of decoder operations. When sampling along a path produces an instruction that prematurely halts execution, it is discarded in favor of other paths. As paths are assessed, top scoring samples are kept as decoder candidates.

Paragraphs are built by sampling words from the language model based on their observed frequency. Each sentence is generated from left to right such that words are added to a sentence only when they have also been observed in the sample text following the combination of words already chosen for the sentence. Retrospection is, however, limited. Since we use an smoothed $n$-gram language model and our maximum $n$-gram length is 5[3], a candidate word $w_5$ will only follow $w_1w_2w_3w_4$ if $w_1w_2w_3w_4w_5$ exists in the training corpus. In more traditional language generation applications, we might perform a random walk through the model, choosing each candidate word at random based on its probability (e.g., if $w_5$ follows $w_1w_2w_3w_4$ with a probability of 0.9, then it would be generated with a probability of 0.9).

While sampling a string from a language model, a traditional language generation application may only be influenced by the probability distribution for each candidate word. Since we are also interested in a word's contribution to execution, we seek a path through the model that maximizes English probability and correct execution behavior simultaneously. To do so, we traverse the language model using the *Viterbi algorithm* [24].

Viterbi is used to reconstruct the most probable sequence of states in a *hidden Markov model*. A hidden Markov model (HMM) is simply a Markov model in which each state is comprised of known parameters (e.g., a word) and unknown parameters (e.g., a word's contribution to our execution goals).

Throughout this process, an objective assessment function scores candidate execution so that we can quantitatively compare candidates. Each decoder instruction is assigned a level number $i$ such that $1 \leq i \leq n$, whereby level $i$ denotes the $i^{th}$ instruction. For each sampled string, the score then indicates the number of desirable instructions it achieves. At the beginning of language generation, we say that the (yet to be generated) instance has a score of zero. For each candidate word, we concatenate it to the string along the path of generation and then execute the string in a sandbox (see Section 6). If the string fails execution (e.g., crashes the simulator), it is discarded. If the addition of the candidate completes the operation specified by the next level, its score is incremented. If execution does not crash or yield completion of the desired operation, the score does not change. Figure 7 illustrates this behavior.

We sample thousands of strings simultaneously and between each round of candidates, keep only the top $m$ samples. Since we do not know the ideal relationship between execution score and word probability at any intermediate stage, we use a greedy algorithm that maximizes our execution goal first. In other words, we always keep the best $m(= 20,000)$ candidates by highest execution score and use language probability to settle ties. The process continues until a sample reaches the $n^{th}$ level, indicating that an English-based decoder has been found.[4]

While objectives change regularly for the duration of the generation process in response to the completion of prior objectives, some conditions hold throughout. We discourage the selection of candidates that reverse desirable effects achieved previously, overwrite critical data, or execute privileged instructions (i.e., crash) at any stage. Furthermore, we refrain from selecting any candidates that use unpredictable data or set flags unpredictably. For instance, performing an arithmetic operation with an operand whose value is unknown can alter the `EFLAGS` register in way that cannot be predicted a priori. Without these constraints, we would waste effort considering candidates that behave erratically, fail to decode encoded payloads, or ultimately crash.

Once a potential decoder is identified, we can encode arbitrary shellcode. After selecting a payload, we encode the target shellcode by continuing to explore the Viterbi search that generated the decoder. The process for encoding payloads is almost identical to the process we describe in Section 4.2 for finding an English decoder. Instead of monitoring the execution behavior of candidates at each step, the objective assessment function now observes how many target bytes are encoded by each candidate, favoring those that encode more of the payload using fewer words. Interestingly enough, encoding the payload places few restrictions on language generation. This is because the encoded data is non-executable and the first two bytes of each four-byte block are unconstrained (as well as the fourth block while accumulation is incomplete).

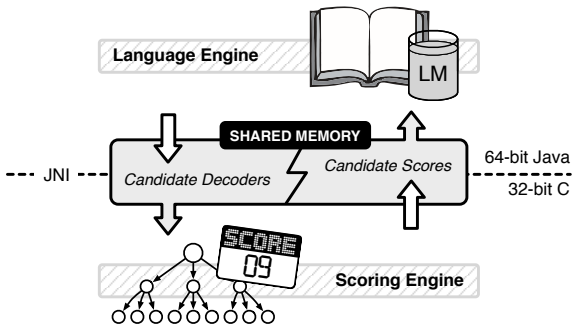


*Figure 8:* Candidate decoders are produced by a language engine and stored in shared memory. Then, they are subsequently executed and evaluated by a scoring engine. As scores are returned to the language engine, candidates are ranked, influencing future candidate selection.

Our implementation is divided into two distinct yet collaborative entities: a language engine and a scoring en-

[3]This value is chosen empirically and represents a trade-off between sampling accuracy and the speed at which samples are generated.

[4]We note, however, that finding a solution with this technique is not guaranteed.

gine. The language engine was constructed in the Java programming language using the *LingPipe* API (see `http://alias-i.com/lingpipe/`). LingPipe is a natural language processing and data mining toolkit that provides an efficient implementation of numerous algorithms and data structures commonly used by computational linguistics applications. We use the toolkit to rapidly build, train, and query our language model.

Two tandem processes comprise the scoring engine. The first process (hereafter referred to as the "executor") executes each candidate decoder while the second (the "watcher") is responsible for controlling and monitoring said execution. The monitor process evaluates candidate behavior (i.e., how it affects the state of the machine) through single-step execution and is implemented using the Linux `ptrace` API. Since our generation technique produces English from left to right, the monitor process favors candidates that perform operations in approximately the same order as our hand-written decoder. This yields the natural scoring mechanism described in Section 5. The scoring engine is also responsible for discouraging the selection of candidates that misbehave, crash, or produce undesirable effects.

The language and scoring engines communicate using shared memory. Communication is facilitated by the *Java Native Interface* (JNI) as depicted in Figure 8. Before generation commences, the JNI component performs a one-time initialization that allocates two shared memory regions: one that holds the potential solution and one that holds its execution results. The JNI component also launches the scoring engine's monitor process. The scoring engine proceeds to evaluate candidates provided by the language engine. The JNI component signals the scoring engine when each new candidate word has been copied into shared memory. Once the signal is received, the scoring engine's execution process fills the stack with random values (to ensure that a solution using uncontrollable stack data is improbable), initializes other registers, and reassigns its instruction pointer to address the candidate decoder. Afterward, the monitor process begins single-step evaluation for the new candidate and provides the language engine with a report of each candidate's score, which helps influence the ongoing Viterbi search and its role in selecting future candidates. This feedback loop ends once the target shellcode has been successfully realized.

## 5.1 An optimized design

An obvious downside of the aforementioned architecture is the use of `ptrace` to single-step the execution of each candidate; indeed, using this approach took 12 hours, on average, to generate a complete decoder. While utilizing `ptrace` turned out to be invaluable in our quest towards automatic generation, its use is ultimately far more inefficient than need be — primarily because it induces multiple context switches between kernel and user space.

One viable alternative to `ptrace` is emulation. That is, instead of using inter-process communication and `ptrace`, we instead emulate the effects of every instruction provided by English as well as the effect of each instruction created by our framework. This is a particularly arduous task because it requires understanding the effects that each English opcode can produce; including the effects to registers and memory locations directly addressed by opcodes as well as the flags register. In addition to being particularly hard to implement, emulation takes a single instruction and can expand it to tens of instructions. Therefore, to avoid using `ptrace` and to eliminate the bulk of the inter-process communication, we use a solution we call *monitored direct execution*.

This optimized design attempts to retain the benefits of direct execution while eliminating the need for `ptrace`. The information required to guide English generation is the same, except we explore more efficient ways to obtain such information. Intuitively, what was formerly two processes is now accomplished by one process that performs both tasks. This is achieved by maintaining two sets of machine state and switching between them to change execution roles. Essentially, both the watcher and the execution "programs" have their own registers, stack, and memory which we call their *state*. They "share" only the memory associated with state switching and the candidate solution.

Switching is accomplished by using two pieces of stub code that saves the state of the watcher program and restores the state of the execution program. Intuitively, we use three separate stacks to minimize the context switch penalty between kernel and user space that arose in the unoptimized case. The first stack is the original watcher stack (indeed, the only "official" stack), the second is an intermediate stack, and the third is the execution program's stack. The intermediate stack houses the information necessary to restore the execution process. The two pieces of stub code mentioned previously use this stack to either save or restore the executor's state. The third stack is simply a portion of memory mapped to a static location. It is saved and restored after each batch of executed instructions, keeping the contents untouched during the watcher's execution.

To boost performance even further, we also forgo single-step execution. A key observation is that we only need information at very specific points in the execution of a candidate solution. Specifically, we only need to know the execution path dictated by changes to the flags register, as well as when memory beyond the current point of candidate execution changes. As such, we pause to inspect execution on two pause conditions. The first is when the execution encounters a jump where `EFLAGS` could be affected by a previous instruction. We identified the conditions under which this could be true by enumerating all the instructions that could change the flags: i.e., all the arithmetic operations (e.g., `inc`, `dec`, `add`, `imul`) and logical operations (e.g., `and`, `or`). (This analysis was made possible because of our earlier use of `ptrace`.) The second pause condition is set in places where we encounter instructions that can change memory; in our implementation, `and` and `add` are the only such instructions. Any operations between either of these pause conditions are executed without intervention.

It is important to note that before the first instructions are executed and during each of the aforementioned pause conditions, the watcher process examines upcoming instructions to avoid either of two undesirable scenarios. Specifically, the execution process cannot be allowed to execute any instructions that (*i*) may result in a crash, e.g., privileged instructions, or (*ii*) result in unpredictable machine state, e.g., by using unpredictable values from registers or memory. Thankfully, we can again take advantage of our experiences using `ptrace` to enumerate and preemptively dismiss any candidate solution containing either of these scenarios.

The result of these optimizations is that we can now routinely generate entire solutions in less than 1 hour on com-

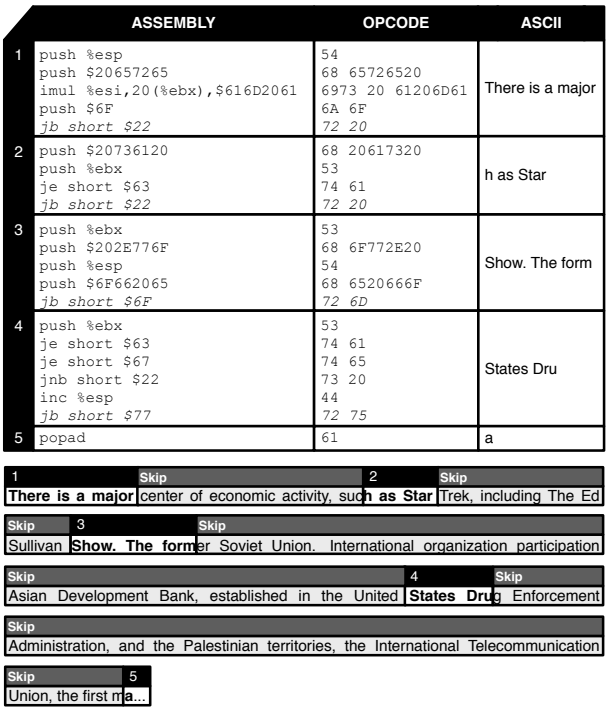


*Figure 9:* Partial anatomy for an excerpt of an automatically generated English encoding.

modity hardware with 4GB of RAM — almost a 12-fold improvement over using `ptrace`.

# 6. EVALUATION

As a preface, we note that given the sensitivity of this work, we *purposely* do not show complete samples of English shellcode. We believe that doing so would be irresponsible, as the risks (i.e., helping the attackers) outweigh the benefits.

Figure 9 shows an annotated excerpt from an English-encoded sample that simply calls `exit(0)`. Notice that the English-encodings we produce generally follow the form and cadence of non-synthetic text. Since our generation engine is merely a proof-of-concept, continued refinement may further reduce the prevalence of seemingly artificial phrases or sentences. The full text is 2054 bytes in length. The segments of text underneath the table with a grey background denote portions of the shellcode that are passed over via jump instructions (and are therefore not executed). In the table, we depict the assembly, machine code, and ASCII-character representations for the bolded (i.e., executed) segments.

Since our focus in this paper is to show that shellcode need not be different in structure than non-executable payload data, we consider assessment of the *quality* of the English we generate outside the scope of this work.[5] Instead, for pedagogical reasons, we revisit a recent approach that is based on using spectrum analysis to detect the presence of executable code in network traffic in lieu of emulation [8]. The key assumption in that work is that the structure of executable content is statistically dissimilar from the structure

[5]Indeed, several conferences (e.g., the International Natural Language Generation Conference) are devoted almost entirely to that topic.

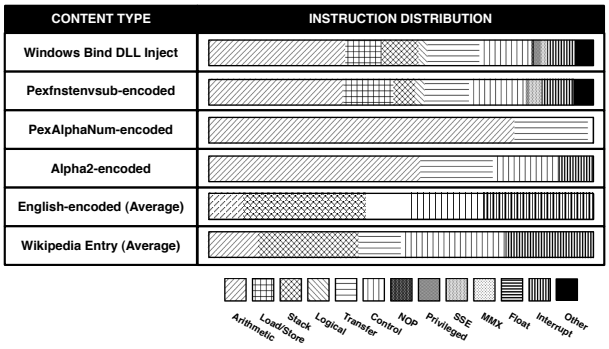


*Figure 10:* Instruction spectrum for various encodings of the `Windows Bind DLL Inject` shellcode included in the Metasploit framework. For comparison, we also "disassembled" 500 Wikipedia articles selected at random and 6 English shellocode samples.

of non-executable data, and so argue that this can be used to identify shellcode. By grouping opcodes into canonical categories (e.g, `push` and `pop` instructions might be classified as stack functions while `and` and `or` might be grouped together with other logical operations), they posit that similar filetypes will have similar categorization patterns. Their results indicate that data and plaintext files have instruction spectrums that are characteristically different from those of executable code[6].

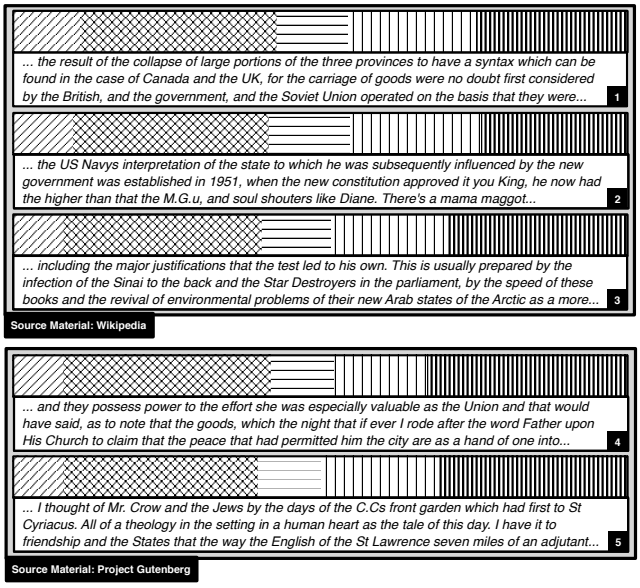


*Figure 11:* Excerpts for alternative encodings of `exit(0)`. The instruction distribution are over the entire shellcode. Refer to the key in Figure 10 for information about instruction categories.

Using the same categories, we classified each `IA32` instruction and produced instruction spectrums for the `Windows Bind DLL Inject` shellcode supplied with the Metasploit framework, various encodings of the shellcode, and Wikipedia articles selected at random. Figure 10 shows the

[6]We note that even using the same 13 instruction groups in [8] we could not verify their results because of inconsistencies and ambiguous statements in their manuscript.

results after sorting each spectrum by category, highlighting the distribution of instruction types in each file. Through visual observation, it is easy to see that the `Pexfnstenvsub` encoding of the Metasploit shellcode is not significantly different than the unmodified shellcode. Alternatively, both alphanumeric encodings are unlike the aforementioned samples and, additionally, have distributions that are most similar to each other.

More importantly, notice that the instruction distribution of the English encoding is most like the instruction distribution of the randomly chosen Wikipedia articles—illuminating the difficulty of distinguishing English shellcode without considering syntactic information. In particular, it is not clear (at least to us), how to easily mitigate this threat without considering the semantics of the input. We show excerpts from other samples we generated in Figure 11.

# 7. CONCLUSION

In this paper we revisit the assumption that shellcode need be fundamentally different in structure than non-executable data. Specifically, we elucidate how one can use natural language generation techniques to produce shellcode that is superficially similar to English prose. We argue that this new development poses significant challenges for inline payload-based inspection (and emulation) as a defensive measure, and also highlights the need for designing more efficient techniques for preventing shellcode injection attacks altogether.

# 8. ACKNOWLEDGEMENTS

We thank Bryan Hoffman, Ryan MacArthur, Scott Coull, John McHugh, Ryan Gardner, and Charles Wright for many insightful discussions. Thank you also to our anonymous reviewers for their invaluable comments and suggestions. This work was funded in part by NFS grants CNS-0627611 and CNS-0852649.

# 9. REFERENCES

[1] P. Akritidis, E. P. Markatos, M. Polychronakis, and K. Anagnostakis. STRIDE: Polymorphic Sled Detection through Instruction Sequence Analysis. In *Proceedings of the International Information Security Conference*, 2005.

[2] K. Borders, A. Prakash, and M. Zielinski. Spector: Automatically Analyzing Shell Code. *Proceedings of the Annual Computer Security Applications Conference*, pages 501–514, 2007.

[3] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *Proceedings of ACM Conference on Computer and Communications Security*, Oct. 2008.

[4] C. Cowan, C. Pu, D. Maier, H. Hinton, 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*, pages 63–78, 1998.

[5] T. Detristan, T. Ulenspiegel, Y. Malcom, and M. S. V. Underduk. Polymorphic Shellcode Engine Using Spectrum Analysis. *Phrack*, 11(61), August 2003.

[6] T. Durden. Bypassing PaX ASLR Protection. *Phrack*, 11(59), July 2002.

[7] K2. ADMmutate. See `http://www.ktwo.ca/c/ADMmutate-0.8.4.tar.gz`.

[8] I. Kim, K. Kang, Y. Choi, D. Kim, J. Oh, and K. Han. A Practical Approach for Detecting Executable Codes in Network Traffic. In *Asia-Pacific Network Operations and Management Symposium*, 2007.

[9] O. Kolesnikov, D. Dagon, and W. Lee. Advanced Polymorphic Worms: Evading IDS by Blending in with Normal Traffic. Technical Report GIT-CC-05-09, Georgia Institute of Technology, 2005.

[10] G. MacManus and M. Sutton. Punk Ode: Hiding Shellcode in Plain Sight. In *Black Hat USA*, 2006.

[11] Obscou. Building IA32 Unicode-Proof Shellcodes. *Phrack*, 11(61), August 2003.

[12] A. One. Smashing The Stack For Fun And Profit. *Phrack*, 7(49), November 1996.

[13] A. Pasupulati, J. Coit, K. Levitt, S. F. Wu, S. H. Li, R. C. Kuo, and K. P. Fan. Buttercup: on Network-based Detection of Polymorphic Buffer Overflow Vulnerabilities. In *IEEE/IFIP Network Operation and Management Symposium*, pages 235–248, May 2004.

[14] U. Payer, P. Teufl, and M. Lamberger. Hybrid Engine for Polymorphic Shellcode Detection. In *Proceedings of Detection of Intrusions and Malware & Vulnerability Assessment*, pages 19–31, 2005.

[15] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Network-level Polymorphic Shellcode Detection using Emulation. In *Proceedings of Detection of Intrusions and Malware & Vulnerability Assessment*, pages 54–73, 2006.

[16] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Emulation-based Detection of Non-self-contained Polymorphic Shellcode. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, 2007.

[17] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. An Empirical Study of Real-world Polymorphic Code Injection Attacks. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats*, 2009.

[18] Rix. Writing IA32 Alphanumeric Shellcode. *Phrack*, 11(57), August 2001.

[19] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of ACM Conference on Computer and Communications Security*, pages 298–307, Oct. 2004.

[20] Y. Song, M. E. Locasto, A. Stavrou, A. D. Keromytis, and S. J. Stolfo. On the Infeasibility of Modeling Polymorphic Shellcode. In *Proceedings of ACM Conference on Computer and Communications Security*, pages 541–551, 2007.

[21] A. Sotirov and M. Dowd. Bypassing Browser Memory Protections. In *Black Hat USA*, 2008.

[22] A. N. Sovarel, D. Evans, and N. Paul. Where's the FEEB? On the Effectiveness of Instruction Set Randomization. In *Proceedings of the USENIX Security Symposium*, 2005.

[23] T. Toth and C. Kruegel. Accurate Buffer Overflow Detection via Abstract Payload Execution. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, pages 274–291, 2002.

[24] A. J. Viterbi. Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269, April 1967.

[25] T. Wana. Writing UTF-8 compatible shellcodes. *Phrack*, 11(62), July 2004.

[26] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. STILL: Exploit Code Detection via Static Taint and Initialization Analyses. *Proceedings of the Annual Computer Security Applications Conference*, pages 289–298, December 2008.

[27] Q. Zhang, D. S. Reeves, P. Ning, and S. P. Iyer. Analyzing Network Traffic to Detect Self-decrypting Exploit Code. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, 2007.