On Deriving Unknown Vulnerabilities from Zero-Day Polymorphic and Metamorphic Worm Exploits

Jedidiah R. Crandall
Dept. Comp. Sci.
Univ. of Calif., Davis
Davis, CA 95616
crandall@cs.ucdavis.edu

Zhendong Su
Dept. Comp. Sci.
Univ. of Calif., Davis
Davis, CA 95616
su@cs.ucdavis.edu

S. Felix Wu
Dept. Comp. Sci.
Univ. of Calif., Davis
Davis, CA 95616
wu@cs.ucdavis.edu

Frederic T. Chong
Dept. Comp. Sci.
Univ. of Calif., Santa Barbara
Santa Barbara, CA 93106
chong@cs.ucsb.edu

ABSTRACT

Vulnerabilities that allow worms to hijack the control flow of each host that they spread to are typically discovered months before the worm outbreak, but are also typically discovered by third party researchers. A determined attacker could discover vulnerabilities as easily and create zero-day worms for vulnerabilities unknown to network defenses. It is important for an analysis tool to be able to generalize from a new exploit observed and derive protection for the vulnerability.

Many researchers have observed that certain predicates of the exploit vector must be present for the exploit to work and that therefore these predicates place a limit on the amount of polymorphism and metamorphism available to the attacker. We formalize this idea and subject it to quantitative analysis with a symbolic execution tool called DACODA. Using DACODA we provide an empirical analysis of 14 exploits (seven of them actual worms or attacks from the Internet, caught by Minos with no prior knowledge of the vulnerabilities and no false positives observed over a period of six months) for four operating systems.

Evaluation of our results in the light of these two models leads us to conclude that 1) single contiguous byte string signatures are not effective for content filtering, and token-based byte string signatures composed of smaller substrings are only semantically rich enough to be effective for content filtering if the vulnerability lies in a part of a protocol that is not commonly used, and that 2) practical exploit analysis must account for multiple processes, multithreading, and kernel processing of network data necessitating a focus on primitives instead of vulnerabilities.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection–*invasive software*

General Terms

Security, languages

Keywords

worms, polymorphic worms, symbolic execution, polymorphism, metamorphism, honeypots

1. INTRODUCTION

Zero-day worms that exploit unknown vulnerabilities are a very real threat. Typically vulnerabilities are discovered by "white hat" hackers using fuzz testing [26, 27], reverse engineering, or source code analysis and then the software vendors are notified. The same techniques for discovering these vulnerabilities could be as easily employed by "black hat" hackers, especially now that computer criminals are increasingly seeking profit rather than mischief. None of the 14 exploits analyzed in this paper are for vulnerabilities discovered by the vendors of the software being attacked. A vulnerability gives the attacker an important primitive (a *primitive* is an ability the attacker has, such as the ability to write an arbitrary value to an arbitrary location in a process' address space), and then the attacker can build different exploits using this primitive.

The host contains information about the vulnerability and primitive that cannot be determined from network traffic alone. It is impossible to generalize how the attack might morph in the future without this information. In order to respond effectively during an incipient worm outbreak, an automated analysis tool must be able to generalize one instance of an exploit and derive protection for the exploited vulnerability, since a worm can build multiple exploits for the same vulnerability from primitives.

1.1 The Need to Be Vulnerability-Specific

If a honeypot or network technology generated an exploit-specific signature for every exploit, the worm author could trivially subvert content filtering by generating a new exploit for each infection attempt. One approach to ameliorate this is to compare multiple exploits and find common substrings. This can be done in the network [21,37] or from TCP dumps of different honeypots [24]. Our results in Section 4 show that contiguous byte string signatures are not semantically rich enough for effective content filtering of polymorphic and metamorphic worms. The same conclusion was reached by Newsome et. al. [28], in which three new kinds of byte-string signatures were proposed that are sets composed of tokens (substrings). For more information see Section 1.3.1. In this paper we generate these tokens for 14 remote exploits using DACODA and conclude that even token-based byte strings are only semantically rich enough to distinguish between worms and valid traffic if the worm exploits a vulnerability that is not found in a commonly used part of a protocol. For example, the signature token "`\x0d\nTransfer-Encoding:\x20chunked\x0d\n\x0d\n`" would have stopped the Scalper worm but also would have dropped valid traffic if valid traffic commonly used chunked encodings. This is the only token for this particular exploit that distinguishes it from ordinary HTTP traffic.

1.2 DACODA: The Da*vis* *Mal*cod*e* A*nalyzer*

Complicating the problem of deriving the vulnerability from a single exploit is the fact that many exploits can involve more than one network connection, multiple processes, multithreading, and a significant amount of processing of network data in the kernel. Such experiences with real exploits have motivated us to develop two different models in order to be more perspicuous in discussing polymorphism and metamorphism: the *Epsilon-Gamma-Pi (ϵ, γ, π)* model [14] for control flow hijacking attacks and the *PD-Requires-Provides* model for exploits. Both of these models take a "from-the-architecture-up" view of the system in which context switches and interprocess communication are simply physical transfers of data in registers and memory.

We have developed a tool called DACODA that analyzes attacks using full-system symbolic execution [22] on every machine instruction. In this paper, we use DACODA for a detailed, quantitative analysis of 14 real exploits. DACODA tracks data from the attacker's network packets to the hijacking of control flow and discovers strong, explicit equality predicates about the exploit vector; *strong, explicit equality predicates* are predicates that show equality between labeled data and an integer that are due to an explicit equality check by the protocol implementation on the attacked machine using a comparison instruction followed by a conditional instruction (typically a conditional jump). Using Minos [13] as an oracle for catching attacks, DACODA honeypots have been analyzing attacks exploiting vulnerabilities unknown to Minos or DACODA with zero observed false positives for the past six months. More details on DACODA's operation are in Section 3.

1.3 Related Work

The details of the Epsilon-Gamma-Pi model are in another paper [14] and will be summarized in Section 2. For categorizing related work in this section we will only state here that, in simple terms, ϵ maps the exploit vector from the attacker's network packets onto the trace of the machine being attacked before control flow hijacking occurs, γ maps the bogus control data used for hijacking control flow (such as the bogus return pointer in a stack-based buffer overflow attack), and π maps the payload executed after control flow has been hijacked.

1.3.1 Vulnerability Specificity

Vigilante [10, 11] captures worms with a mechanism similar to Minos [13], but based on binary rewriting of a single process, and uses dynamic dataflow analysis to generate a vulnerability signature. The basic idea proposed in Costa et al. [10] is to replay the execution with an increasingly larger suffix of the log and check for the error condition. Binary rewriting of a single process does not capture interprocess communication, inter-thread communication, or any data processing that occurs in kernel space. It also modifies the address space of the process being analyzed, which has the potential of breaking the exploit in its early stages [3]. DACODA is a full-system implementation and does not modify the system being analyzed. Another important distinction of DACODA is that, because it is based on the Epsilon-Gamma-Pi model, DACODA's symbolic execution helps distinguish between what data looks like on the network and what it looks like at various stages of processing on the host. Encodings such as UNICODE encodings or string to integer conversion cannot be captured by simply comparing I/O logs to TCP dumps.

TaintCheck [29] is also based on binary rewriting of a single process and proposed dynamic slicing techniques as future work to generate vulnerability-specific signatures. DACODA is based on symbolic execution of every machine instruction in the entire system. For RIFLE [41] an Itanium architecture simulator was augmented with dataflow analysis capabilities similar to DACODA, without predicate discovery, but the aim was to enforce confidentiality policies while DACODA's aim is to analyze worm exploits.

Newsome et. al. [28] proposed three types of signatures based on tokens. These tokens can be ordered or associated with scores. Polygraph, unlike EarlyBird [37], Autograph [21], or Honeycomb [24], does not automatically capture worms but instead relies on a flow classifier to sort worm traffic from benign traffic with reasonable accuracy. The invariant bytes used for tokens were typically from either protocol framing (ϵ) or the bogus control data (γ). It was suggested that the combination of these could produce a signature with a good false positive and false negative rate. Protocol framing describes a valid part of a protocol, such as "`HTTP GET`" in HTTP. Also, γ permits too much polymorphism according to our analysis of exploits caught by Minos honeypots [14], due to register springs.

Register springs are a technique whereby the bogus function pointer or return pointer overwritten by a buffer overflow points to an instruction in a library (or the static program) that is a jump or call to a register pointing into the buffer containing π. Newsome et. al. [28] correctly states that for register springs to be stable the address must be common across multiple Windows versions and cites Code Red as an example, but Code Red and Code Red II used an address which was only effective for Windows 2000 with Service Pack 1 or no service packs (the instruction at 0x7801cbd3 disassembles to "`CALL EBX`" only for msvcrt.dll version 6.10.8637 [33]). Even with this limitation Code Red and Code Red II were successful by worm standards, so the hundreds or sometimes thousands of possible register springs typical of Windows exploits cannot be ignored.

One current limitation of DACODA is performance. Our Bochs-based implementation of DACODA achieves on the order of hundreds of thousands of instructions per second on a 3.0 GHz Pentium 4 with an 800 MHz front side bus. Memory bandwidth is the limiting factor, and DACODA barely achieves good enough performance to be infected by a worm on a 2.8 GHz Pentium 4 with a 533 MHz front side bus. All that really is required to detect the attack is Minos [13], which would have virtually no overhead in a hardware implementation and could possibly have performance within an order of magnitude of native execution if implemented on a higher performance emulator such as QEMU [51]. After Minos detected an attack DACODA could be invoked by replaying either the TCP traffic [19,20] or the entire attack trace [16].

1.3.2 *Modeling Polymorphism*

Ideas similar to our PD-Requires-Provides model for exploit polymorphism and metamorphism are presented in Rubin et. al. [34,35]. The PD-Requires-Provides model is at a much lower level of abstraction. Rubin et. al. [34,35] do not distinguish between what the exploit looks like on the network and what it looks like when it is processed on the host, as our Epsilon-Gamma-Pi model does. These works were also intended for generating exploits based on known vulnerabilities and not for analyzing zero-day exploits to derive protection for unknown vulnerabilities. A more recent work [44] generates vulnerability-specific signatures for unknown exploits but requires a detailed specification of the protocol that the exploit uses (such as SMB or HTTP). DACODA needs no specification because of symbolic execution, at the cost of not having a full specification against which to model check signatures.

In Dreger et. al. [15] host-based context was used to enhance the accuracy of network-based intrusion detection but this was done from within the Apache HTTP server application. Ptacek and Newsham [32] cover some of the same ideas as we do but within the context of network evasion of network intrusion detection systems. Christodorescu and Jha [7] looked at polymorphism of viruses with examples from real viruses, but polymorphic virus detection and polymorphic worm detection are two different problems; a worm needs to be able to hijack control flow of remote hosts because worms use the network as their main medium of infection.

1.3.3 *Polymorphic Worm Detection*

Many researchers have studied polymorphic techniques and detection mechanisms in π [1,6,8,23,25,31,40]. Several of the mechanisms which have been proposed are based on the existence of a NOP sled which simply is not applicable to Windows exploits, nearly all of which use register springs [14]. The executable code itself could be made polymorphic and metamorphic with respect to probably any signature scheme if we are to consider the relatively long history of polymorphic computer viruses [38]. Other works have focused exclusively on γ [30] which can be polymorphic because there are usually hundreds or even thousands of different register springs an attacker might use [14]. We have argued in another paper [14] that π and γ permit too much polymorphism, motivating a closer look at ϵ instead. The focus of this paper is on polymorphism and metamorphism of ϵ. Other papers have focused on ϵ [10,28,34,35,43,44], all of which have already been discussed in this section except for Shield [43]. Shields are a host-based solution which are an alternative to patches. They are vulnerability-specific but only for known vulnerabilities.

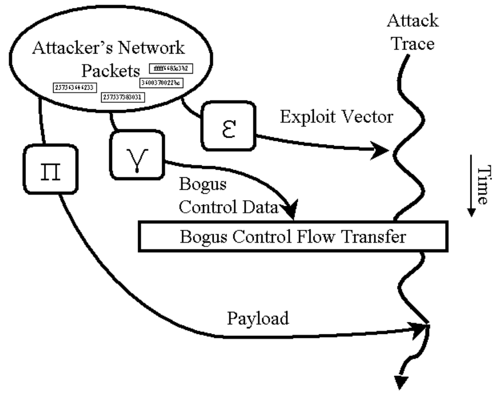

Figure 1: The Epsilon-Gamma-Pi Model.

1.3.4 *Our Main Contributions*

The main distinction of our work is that we focus on unknown vulnerabilities and use models based on our experience with analyzing 14 real exploits to give a detailed and quantitative analysis of polymorphism and metamorphism for the exploit vector mapped by ϵ. Our main contributions are 1) a tool for whole-system symbolic execution of remote exploits, 2) quantitative data on the amount of polymorphism available in ϵ for 14 actual exploits, which also shows the importance of whole-system analysis, and 3) a model for understanding polymorphism and metamorphism of ϵ. Actual generation of vulnerability-specific signatures with low false positive and false negative rates is left for future work.

1.4 Structure of the Paper

The rest of the paper is structured as follows. Section 2 summarizes the Epsilon-Gamma-Pi model for control flow hijacking attacks from past work [14], followed by Section 3, which details how DACODA generated the results from analyzing real exploits that are in Section 4. The PD-Requires-Provides model is described in Section 5 to help understand polymorphism and metamorphism. After discussing future work in Section 6, we give our conclusions about byte string signature schemes and host-based semantic analysis.

2. THE EPSILON-GAMMA-PI MODEL

The Epsilon-Gamma-Pi model [14] is a model of control flow hijacking attacks based on projecting the attacker's network packets onto the trace of the machine being attacked. The *row space of a projection* is the *network data* that is relevant to that projection, while the *range of a projection* is the physical data *used by the attacked machine for control flow decisions*. The Epsilon-Gamma-Pi model can avoid confusion when, for example, the row space of γ for Code Red II is UNICODE encoded as "`0x25 0x75 0x62 0x63 0x64 0x33 0x25 0x75 0x37 0x38 0x30 0x31`" coming over the network but stored in little-endian format in the range of γ as the actual bogus Structured Exception Handling

(SEH) pointer "`0xd3 0xcb 0x01 0x78`". These encodings of 0x7801cbd3 are captured by γ.

The mappings of a particular exploit are chosen by the attacker but constrained by the protocol as implemented on the attacked machine. A single projection is specific to an exploit, not to a vulnerability. A vulnerability can be thought of as a set of projections for ϵ that will lead to control flow hijacking, but the term vulnerability may be too subjective to define formally. Sometimes vulnerabilities are a combination of program errors, such as the ASN.1 Library Length Heap Overflow vulnerability [52, bid 9633] which was a combination of two different integer overflows. We can say that a system is vulnerable to a remote control flow hijacking attack if there exists any combination of IP packets that cause bogus control flow transfer to occur.

The projection ϵ maps network data onto control flow decisions before control flow hijacking takes place, while γ maps the bogus control data itself during control flow hijacking and π typically maps the attacker's payload code that is directly executed after control flow is hijacked. In simple terms, ϵ maps the exploit vector, γ maps the bogus control data, and π maps the payload code as illustrated by Figure 1.

2.1 Polymorphism and Metamorphism

The Epsilon-Gamma-Pi model also provides useful abstractions for understanding polymorphism and metamorphism. Worm signature generation with any particular technique can be seen as a characterization of one or more of the three mappings possibly combined with information about the attack trace on the infected host. Polymorphism and metamorphism seek to prevent this characterization from enabling the worm defense to distinguish the worm from other traffic as it moves over the network. In the extreme the attacker must, for different infections, change these three mappings and the attack trace on the infected machine enough so that knowledge about the attack trace and characterizations of the three mappings cannot permit identification of the worm with a low enough error rate to stop the worm from attaining its objective. In practice, however, the benefit of surprise goes to the attacker, and polymorphism and metamorphism will be with respect to some specific detection mechanism that has actually been deployed. Polymorphism changes bytes in the row spaces of the three projections without changing the mappings, while metamorphism uses different mappings each time. Unless otherwise stated, in this paper a signature is a set of byte strings (possibly ordered) that identify the worm, and polymorphism and metamorphism are with respect to this set of strings. The Epsilon-Gamma-Pi model is more general than byte string signatures, however. One of the main results of this paper is that simple byte string matching, even for sets of small strings or regular expressions, can be inadequate for worm content filtering for realistic vulnerabilities.

2.2 Motivation for the Model

The Epsilon-Gamma-Pi model is general enough to handle realistic attacks that do not follow the usual procession of opening a TCP connection, adhering to some protocol through the exploit vector phase until control flow is hijacked, and then executing the payload in the thread that was exploited. IP packets in the Epsilon-Gamma-Pi model and in the DACODA implementation are raw data subject to interpretation by the host, since "information only has meaning in that it is subject to interpretation" [9], a fact that is at the heart of understanding viruses and worms. An attacker might use an arbitrary write primitive in one thread to hijack the control flow of another, or hijack the control flow of the thread of a legitimate user.

Using symbolic execution, DACODA is able to discover strong, explicit equality predicates about ϵ. Specifically, DACODA discovers the mapping ϵ and also can use control flow decisions predicated explicitly on values from the range of ϵ to discover predicates about the bytes of network traffic from which the values were projected (the row space of ϵ). These predicates can be used for signature generation, but in this paper we use DACODA to characterize ϵ quantitatively for a wide variety of exploits. This quantitative analysis plus our experiences with analyzing actual exploit vectors serve as a guide towards future work in this area.

For all three projections, DACODA tracks the data flow of individual bytes from the network packets to any point of interest. Thus it also is helpful in answering queries about where the payload code comes from or how the bogus control data is encoded within the network traffic.

2.3 The Need for an Oracle

To distinguish ϵ, γ, and π, and also to provide the analysis in a timely manner, DACODA needs an oracle to raise an alert when bogus control flow transfer has occurred. For the current implementation we use Minos [13] as an oracle to catch low-level control data attacks. Minos is basically based on taintchecking to detect when data from the network is used as control data. Thus it does not catch attacks that hijack control flow at a higher level abstraction than low-level execution, such as the Santy worm or the attacks described in Chen et. al. [5], but DACODA is equally applicable to any control flow hijacking attack. For example, in an attack where the filename of a file to be executed, such as "`/usr/bin/counterscript`", is overwritten with "`/bin/sh`" then executed yielding a shell, ϵ would map the exploit vector leading to the overwrite, γ would map the string "`/bin/sh`", and π would map the commands executed once the shell was obtained. Minos will not catch this attack but DACODA will still provide an analysis given the proper oracle. Any worm that spreads from host to host must hijack control flow of each host at one level of abstraction or another.

3. HOW DACODA WORKS

DACODA is currently being emulated in a full-system Pentium environment based on the Bochs emulator [46]. When a network packet is read from the Ethernet device every byte of the packet is labeled with a unique integer. Reading the packet off the Ethernet is the last chance to see all bytes of the packet intact and in order, because the NE2000 driver often reads parts of packets out of order.

During its lifetime this labeled data will be stored in the NE2000 device's memory pages, read into the processor through port I/O, and moved and used in computation by various kernel- and user-space threads and processes. DACODA will track the data through all of this and discover equality predicates every time the labeled data or a symbolic expression is explicitly used in a conditional control flow transfer. Symbolic execution occurs in real-time so that when an oracle (Minos [13] in the current implementation)

Explanation	C++-like Pseudo-code

MakeNewQuadMem() is used for reading four bytes of memory and making a QuadExpression from them, unless we find that the memory word already contains a QuadExpression.	Expression *MakeNewQuadMem(Addr)
	    FirstByte = ReadMemByteExpr(Addr);
	    if (FirstByte→IsAQuadExpr()) return FirstByte;
	    else return new QuadExpr(
	        ReadMemByteExpr(Addr + 0)
	      , ReadMemByteExpr(Addr + 1)
	      , ReadMemByteExpr(Addr + 2)
	      , ReadMemByteExpr(Addr + 3));

MakeNewQuadRegister() is the same as MakeNewQuadMem() but for 32-bit register reads.	Expression *MakeNewQuadRegister(Index)
	    FirstByte = ReadRegisterByteExpr(Index, 0);
	    if (FirstByte→IsAQuadExpr()) return FirstByte;
	    else return new QuadExpr(
	        ReadRegisterByteExpr(Index, 0)
	      , ReadRegisterByteExpr(Index, 1)
	      , ReadRegisterByteExpr(Index, 2)
	      , ReadRegisterByteExpr(Index, 3));

WriteQuadMem() stores a QuadExpression in a way that MakeNewQuadMem() can find it.	void WriteQuadMem(Addr, Expr)
	    WriteMemByteExpr(Addr + 0, Expr);
	    WriteMemByteExpr(Addr + 1, NULL);
	    WriteMemByteExpr(Addr + 2, NULL);
	    WriteMemByteExpr(Addr + 3, NULL);

WriteQuadRegister() is the same as WriteQuadMem() but for 32-bit register writes.	void WriteQuadRegister(Index, Expr)
	    WriteRegisterByteExpr(Index, 0, Expr);
	    WriteRegisterByteExpr(Index, 1, NULL);
	    WriteRegisterByteExpr(Index, 2, NULL);
	    WriteRegisterByteExpr(Index, 3, NULL);

MakeNewQuadConstant() simply uses bit masks and shifts to split the 32-bit constant into 4 8-bit constants.	Expression *MakeNewQuadConstant(0xAABBCCDD)
	    return new QuadExpression(
	        new Constant(0xAA)
	      , new Constant(0xBB)
	      , new Constant(0xCC)
	      , new Constant(0xDD));

Table 1: How QuadExpressions are Handled.

determines that control flow has been hijacked, DACODA simply summarizes the results of its analysis.

As an example, suppose a byte of network traffic is labeled with "`Label 1832`" when it is read from the Ethernet card. This label will follow the byte through the NE2000 device into the processor where the kernel reads it into a buffer. Suppose the kernel copies this byte into user space and a user process moves it into the AL register, adds the integer 4 to it, and makes a control flow transfer predicated on the result being equal to 10.

```
mov     al,[AddressWithLabel1832]
     ; AL.expr <- (Label 1832)
add     al,4
     ; AL.expr <- (ADD AL.expr 4)
     ; /* AL.expr == (ADD (LABEL 1832) 4) */
cmp     al,10
     ; ZFLAG.left <- AL.expr
     ; /* ZFLAG.left == (ADD (Label 1832) 4) */
     ; ZFLAG.right <- 10
je      JumpTargetIfEqualToTen
     ; P <- new Predicate(EQUAL ZFLAG.left ZFLAG.right)
     ; /* P == (EQUAL (ADD (Label 1832) 4) 10) */
     ; if (ZF == 1) AddToSetOfKnownPredicates(P);
     ; /* Discover predicate if equality branch taken */
```

This illustrates how DACODA will discover the predicate (in prefix notation), "`(EQUAL (ADD (Label 1832) 4) 10)`". This predicate from the range of ϵ can be used to infer a predicate about the row space of ϵ: that the byte that was labeled with "`Label 1832`" is equal to 6.

For 16- or 32-bit operations DACODA concatenates the labels for two or four bytes into a *DoubleExpression* or a *QuadExpression*, respectively. We define a *strong, explicit equality predicate* to be an equality predicate that is exposed because of an explicit check for equality. Thus a comparison of an unsigned integer that yields the predicate that the integer is less than 1 is not explicit and will not be discovered by DACODA (though it implies that this integer is equal to 0).

DACODA also discovers equality predicates when a labeled byte or symbolic expression is used as a jump or call target, which is common in code compiled for C switch statements and is how DACODA is able to detect important predicates such as the first data byte in the UDP packet of the Slammer worm, "`0x04`", the only real signature this attack has. When a symbolic expression is used in an address for an 8- or 16-bit load or store operation the address becomes part of the symbolic expression of the value loaded or stored (a *Lookup* expression is created which encapsulates both the value and the address used to load or store it). This type of information flow is important for tracking operations such as the ASCII to UNICODE conversion of Code Red II.

There are six kinds of expressions: *Labels*, *Constants*, *DoubleExpressions*, *QuadExpressions*, *Lookups*, and *Operations*. Every byte of the main physical memory, the general purpose registers, and the NE2000 card's memory are associated with an expression, which can be NULL. The Zero Flag (ZF) is used by the Pentium for indicating equality or inequality. We associate two expressions with ZF, left and right, to store the expressions for the last two data that were compared. ZF can also be set by various arithmetic instructions but only explicit comparison instructions set the left and right pointers in our implementation. These pointers become an equality predicate if *any* instruction subsequently checks ZF and finds it to be set.

Table 2 summarizes all of the various rules about how DACODA propagates expressions and discovers predicates.

Explanation	Assembly Example	What DACODA Does in C++-like Pseudo-code

Moves from register to memory, memory to register, or register to register just copy the expressions for the bytes moved. The same applies to PUSHEs and POPs.	mov edx,[ECX]
	
	mov al,bh
	
	mov [EBP+10],cl
		WriteRegisterByteExpr(INDEXOFEDX, 0, ReadMemByteExpr(ecx+0));
		WriteRegisterByteExpr(INDEXOFEDX, 1, ReadMemByteExpr(ecx+1));
		WriteRegisterByteExpr(INDEXOFEDX, 2, ReadMemByteExpr(ecx+2));
		WriteRegisterByteExpr(INDEXOFEDX, 3, ReadMemByteExpr(ecx+3));
		
		WriteRegisterByteExpr(INDEXOFEAX, 0,
		ReadRegisterByteExpr(INDEXOFEBX, 1));
		
		WriteMemByteExpr(ebp+10, ReadRegisterByteExpr(INDEXOFECX, 0));

8- and 16-bit lookups carry their addresses with them. Without this the 0x7801cbd3 bogus SEH pointer of Code Red II would have no expression.	mov dx,[ECX]	DoubleExprFromMem = MakeNewDoubleMem(ecx);
		AddrResolved = MakeNewDoubleRegister(INDEXOFECX);
		ExprForDX = new Lookup(AddrResolved, DoubleExprFromMem);
		WriteDoubleRegister(INDEXOFEDX, ExprForDX);

Jumps or calls to addresses that have non-NULL expressions imply an equality predicate on that expression; needed
 	
for Slammer.
	mov edx,[EBP+fffffbf4]
	
	jmp [42cfa23b+EDX<<2]
		ExprForEDX = MakeNewQuadMem(ebp+0xfffffbf4);
		WriteQuadRegister(INDEXOFEDX, ExprForEDX);
		
		AddrResolved = new Operation("ADD",
		MakeNewQuadConstant(0x42cfa23b),
		new Operation("SHR", MakeNewQuadRegister(INDEXOFEDX),
		new Constant(2)));
		AddToListOfKnownPredicates("EQUAL", AddrResolved,
		MakeNewQuadConstant(0x42cfa23b+edx<<2));

Strong, explicit equality predicates are discovered when a CMP, CMPS, SCAS, or TEST instruction is followed by any instruction that checks the Zero Flag (ZF) and ZF indicates equality. Examples are conditional equality jumps such as JE, conditional moves, or "REP SCAS".	cmp edx,[ESI]
	
	je 7123abcd
		ZFLAG.left = MakeNewQuadRegister(INDEXOFEDX);
		ZFLAG.right = MakeNewQuadMem(esi);
		if ((ZFLAG.right != NULL) &&
		(ZFLAG.left == NULL)) ZFLAG.left = new Constant(edx);
		if ((ZFLAG.left != NULL) &&
		(ZFLAG.right == NULL)) ZFLAG.right = new Constant([esi]);
		
		P = new Predicate("EQUAL", ZFLAG.Left, ZFLAG.Right);
		if (ZF == 1 && ((ZFLAG.Left != NULL) || (ZFLAG.Right != NULL)))
		AddToListOfKnownPredicates(P);

Operations such as ADDs, other arithmetic operations, bit shifts, or logical bit operations simply create a new Operation expression which can be written into the slot for QuadExpressions and will be encapsulated as a QuadExpression the next time it is read. The same applies to DoubleExpressions, and 8-bit operations are straightforward.	add eax,[EBX] shr eax,3 mov [ECX],eax	WriteQuadRegister(INDEXOFEAX, new Operation("ADD", MakeNewQuadRegister(INDEXOFEAX), MakeNewQuadMem(ebx)); WriteQuadRegister(INDEXOFEAX, new Operation("SHR", MakeNewQuadRegister(INDEXOFEAX), new Constant(3)); WriteQuadMem(ecx, MakeNewQuadRegister(INDEXOFEAX));

Table 2: Special Rules and Example Instructions.

Exploit	OS	Port(s)	Class	bid [52]	Vulnerability Discovery
LSASS (Sasser)	Windows XP	445 TCP	Buffer Overflow	10108	eEye
DCOM RPC (Blaster)	Windows XP	135 TCP	Buffer Overflow	8205	Last Stage of Delirium
Workstation Service	Windows XP	445 TCP	Buffer Overflow	9011	eEye
RPCSS	Windows Whistler	135 TCP	Buffer Overflow	8459	eEye
SQL Name Resolution (Slammer)	Windows Whistler	1434 UDP	Buffer Overflow	5311	David Litchfield
SQL Authentication	Windows Whistler	1433 TCP	Buffer Overflow	5411	Dave Aitel
Zotob	Windows 2000	445 TCP	Buffer Overflow	14513	Neel Mehta
IIS (Code Red II)	Windows Whistler	80 TCP	Buffer Overflow	2880	eEye
wu-ftpd Format String	RedHat Linux 6.2	21 TCP	Format String	1387	tf8
rpc.statd (Ramen)	RedHat Linux 6.2	111 & 918 TCP	Format String	1480	Daniel Jacobiwitz
innd	RedHat Linux 6.2	119 TCP	Buffer Overflow	1316	Michael Zalewski
Apache Chunk Handling (Scalper)	OpenBSD 3.1	80 TCP	Integer Overflow	5033	N. Mehta, M. Litchfield
ntpd	FreeBSD 4.2	123 TCP	Buffer Overflow	2540	Przemyslaw Frasunek
Turkey ftpd	FreeBSD 4.2	21 TCP	Off-by-one B.O.	2124	Scrippie

Table 3: Exploits Analyzed by DACODA.

Table 1 shows how QuadExpressions are handled. A more straightforward way to handle QuadExpressions would be to place a pointer to the QuadExpression into all four bytes' expressions for that 32-bit word and let the index of each byte determine which of the four bytes in the QuadExpression it should reference, which is how DoubleExpressions are handled. For QuadExpressions, however, this causes numerous performance and memory consumption problems. The scheme in Table 1 is more efficient but may drop some information if, for example, a QuadExpression is written to a register, then a labeled byte is written into a higher order byte of that register, and then the QuadExpression is read from the register. From our experience such cases should be extremely rare, and it would be relatively straightforward to fix but Table 1 is the implementation used to generate the results in Section 4.

4. EXPLOITS ANALYZED BY DACODA

This section will summarize the results produced by DACODA, detail Code Red II as a concrete example, and then enumerate complexities, challenges, and facts worth noting about the exploits analyzed. We adopt the idea of tokens from Polygraph [28] and consider a byte to be tokenizable if DACODA discovers some strong, explicit equality predicate about it.

4.1 Summary

Table 3 summarizes the exploits that DACODA has analyzed. All of the Windows exploits except one (SQL Authentication) were actual attacks or worms from the Internet to DACODA honeypots, while all others were performed by the authors. Identifying the packets involved in each attack was done manually by inspection of the dumped network traffic. Since all packets for each attack were either UDP or TCP we used a summary algorithm that used knowledge of these protocols so that the results could remain more intuitive by not including predicates about the transport layer protocol header, unless they also include labeled bytes from a data field (such as what happens in reverse DNS lookups).

When DACODA discovers a predicate, the Current Privilege Level (CPL) of the processor is checked to determine whether the predicate is discovered while running kernel-space code or while running user-space code. These results are presented in Table 4. The CR3 register in the Pentium is used to index the base of the page table of the current task and is therefore a satisfactory replacement for a process ID (PID). Table 4 also shows the results generated by DACODA as to how many different processes are involved in predicate discovery and are therefore an integral part of understanding the attack. This table includes not only conventional processes but also processes that run only in kernel space such as the Windows SYSTEM process.

Table 5 summarizes the results from preliminary, naive signature generation using DACODA. Note that we make no strong claims as to DACODA's completeness because it is possible that a byte may have a strong equality predicate that is not due to an explicit check for equality. It is also possible that tokens discovered by DACODA are not really invariant for various reasons described later in this section. Also, multiple bytes may be involved with a single predicate and a single byte may be involved with multiple predicates, so there is not a one-to-one relationship between bytes and predicates. Surprisingly, some predicates are repeated such as the "`GET`" token from Code Red II which is checked four times in four different places by the IIS web server. The numbers for predicates and tokens are provided here as an approximation to get a sense of the design space and may vary slightly from the true invariant signatures for these exploits. The format for Table 5 is such that "`3(18)`" means that there are three tokens that are 18 bytes in length.

Validation of the results was done, to the extent possible, by comparing the results to our knowledge of the exploits and the protocols involved.

4.2 Code Red II as a Concrete Example

The UNICODE encoding of the bogus Structured Exception Handling pointer and payload are captured by DACODA's symbolic expressions, as is the fact that the row spaces and ranges of ϵ, γ, and π are not disjoint sets of bytes. DACODA also shows that the exploit vector permits a great deal of polymorphism.

The exploit vector for Code Red II is a GET request:

```
GET /default.ida?XXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX%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\x0d\n.
```

DACODA discovers strong equality predicates for the tokens "`GET\x20/`", "`.ida?`", the UNICODE tokens "`%u`", spaces, new line characters, and "`%u00=a\x20,HTTP/1.0\x0d\n`". Only a single "`%u`" is necessary to cause ASCII to UNICODE conversion and overflow the buffer. The "`.ida`" file can have any filename, real or not, and can also end with "`.idq`". Thus the following is a valid exploit vector for the same vulnerability:

```
GET /notarealfile.idq?UOIRJVFJWPOIVNBUNIVUWIF
OJIVNNZCIVIVIGJBMOMKRNVEWIFUVNVGFWERIOUNVUNWI
UNFOWIFGITTOOWENVJSNVSFDVIRJGOGTNGTOWGTFGPGLK
JFGOIRWTPOIREPTOEIGPOEWKFVVNKFVVSDNVFDSFNKVFK
GTRPOPOGOPIRWOIRNNMSKVFPOSVODIOREOITIGTNJGTBN
VNFDFKLVSPOERFROGDFGKDFGGOTDNKPRJNJIDH%u1234D
SPPOITEBFBWEJFBHREWJFHFRG=bla HTTP/1.0\x0d\n.
```

Though it contains no real bogus control data or payload, it will cause the bogus control flow transfer to occur (from the return pointer, not the SEH pointer in this case). The current DACODA implementation treats all operations as uninterpreted functions so there is one spurious tokenization for this exploit, the one that includes "`00=a`", which should be just "`=`". This is because the "`=`" character is located by bit shifts instead of direct addressing, and DACODA cannot determine that the other three characters are dropped before the explicit equality check without semantic information about the bit shifts. This is the only example of such a problem with uninterpreted functions we discovered.

4.3 Complexities and Challenges

This section discusses some of the facts that must be taken into account when designing an automated worm analysis technique for deriving protection for an unknown vulnerability from a zero-day polymorphic and metamorphic worm exploit.

Exploit Name	**Total Predicates**	**Kernel-space Predicates**	**User-space Predicates**	**Processes Involved**	**Multiple Threads**
LSASS	305	223	82	SYSTEM and lsass.exe	Yes
DCOM RPC	120	0	120	svchost.exe	Yes
Workstation Service	286	181	105	SYSTEM, svchost.exe, ??, ??, and lsass.exe	Yes
RPCSS	38	2	36	SYSTEM and svchost.exe	Yes
SQL Name Res. (Slammer)	1	0	1	SQL Server	Yes
SQL Authentication	7	0	7	SQL Server	Yes
Zotob	271	177	94	SYSTEM, services.exe, and ??	Yes
IIS (Code Red II)	107	0	107	IIS Web Server	No
wu-ftpd Format String	2288	0	2288	wu-ftpd	No
rpc.statd	44	0	44	portmap and rpc.statd	No
innd	329	41	288	innd and nnrpd	No
Apache Chunk Handling	3499	4	3495	httpd	No
ntpd	17	0	17	ntpd	No
Turkey	347	98	249	ftpd	No

Table 4: Where exploits are discovered.

Exploit Name	**Longest Token**	**Token length histogram as "Number(Size in bytes)"**
LSASS	36	1(36),1(34),3(18),2(14),1(12),5(9),5(8),2(5),15(4),2(3),39(2),19(1)
DCOM RPC	92	1(92),1(40),1(20),2(18),1(14),5(8),15(4),2(3),13(2),8(1)
Workstation Service	23	1(23),5(18),1(16),2(14),1(12),4(10),8(8),1(6),5(5),8(4),1(3),42(2),22(1)
RPCSS	18	2(18),2(8),5(4),9(2),8(1)
SQL Name Res. (Slammer)	1	1(1)
SQL Authentication	4	3(4),3(1)
Zotob	36	1(36),1(34),2(18),1(16),1(14),1(12),2(8),3(5),11(4),2(3),32(2),6(1)
IIS (Code Red II)	17	1(17),3(5),23(2),1(1)
wu-ftpd Format String	283	4(283),4(119),4(11),1(10),1(9),1(6),4(5),3(4),4(3),10(2),41(1)
rpc.statd	16	2(16),1(8),2(4),10(2),13(1)
innd	27	1(27),1(21),1(13),1(11),2(10),2(9),2(6),6(5),9(4),12(3)
Apache Chunk Handling	32	1(32),24(13),23(11),1(8),1(6),2(5),1(3),3(2),3(1)
ntpd	8	1(8),2(4),2(2)
Turkey	21	2(21),1(12),2(6),6(5),16(4),23(2),14(1)

Table 5: Signature Tokens.

4.3.1 Processing of Network Data in the Kernel

The most salient feature of the LSASS exploit is the amount of protocol that the attack must traverse through in the kernel itself before it even is able to reach the vulnerable process, *lsass.exe*, through the named pipe "\\`PIPE`\`lsarpc`". For a step-by-step explanation of the LSASS exploit see the eEye advisory [48]. The Windows kernel space contains a great deal of executable code that handles network traffic including Transport Device Interfaces (TDIs), Remote Procedure Calls (RPC), Ancillary Function Driver File System Drivers (AFD FSDs), Named Pipe FSDs, Mailslot FSDs, NetBIOS emulation drivers, and more [36]. Today, even HTTP requests are being processed in the kernel space with a network driver contained in IIS 6.0 [2]. Thus attack analysis must include the kernel.

Furthermore, it is not necessary for a remote exploit to ever involve a user-space process. A remote memory corruption vulnerability in the kernel may allow an attacker to execute arbitrary code directly in "`CPL==0`" (the kernel space). Such an exploit is described by Barnaby Jack [2] that exploits a kernel-space buffer overflow in a popular firewall program. Microsoft recently released an advisory describing a heap corruption vulnerability in the kernel-space SMB driver that could allow remote code execution [50, MS05-027]. Linux and BSD do much less processing of network data in kernel space but are nonetheless susceptible to the same problem [52, bid 11695].

4.3.2 Multiple Processes Involved

The rpc.statd exploit is interesting because it is possible that the vulnerable service, *rpc.statd*, may be listening on a different port for every vulnerable host. This is only probable if the different vulnerable hosts are running different operating system distributions. Nonetheless, the initial connection to *portmap* to find the *rpc.statd* service is an important part of the exploit to analyze.

The innd exploit works by posting a news message to a newsgroup, in this case "`test`", and then canceling that message by posting a cancellation message to the group called "`control`". The buffer overflow occurs when a log message is generated by the *nnrpd* service, which is invoked by the *innd* process, because the e-mail address of the original posting is longer than the buffer reserved for it. In this particular exploit the entire exploit is carried out through a single TCP connection, but it is possible that the attacker could upload the payload and bogus return pointer onto the vulnerable host's hard drive using one TCP connection from one remote host and then invoke the buffer overflow via a different TCP connection coming from a different remote host.

4.3.3 Multithreading and Multiple Ports

In addition to multi-stage attacks like the innd exploit, many Windows services are multithreaded and listen on multiple ports. The SQL Server is multithreaded and listens on ports 1434 UDP and 1433 TCP. The DCOM RPC, Workstation Service, RPCSS, and Zotob exploits have the same property. The Windows Security Bulletin for the LSASS buffer overflow [50, MS04-011] recommends blocking UDP ports 135, 137, 138, and 445, and TCP ports 135, 139, 445, and 593; plus, the *lsass.exe* process is multithreaded meaning that, for example, the payload and the exploit could be introduced into the process' address space simultaneously through two different connections on two different ports. Most exploits allow some form of arbitrary memory corruption such as writing an arbitrary value to an arbitrary location or writing a predictable value to an arbitrary location. Even simple stack-based buffer overflows can have this property, like the RPC DCOM exploit or the Slammer exploit. In Slammer, a certain word just beyond the bogus return pointer can point to any writable address where the value 0 is written just before the bogus control flow transfer occurs.

Any open TCP port 1433 connection can be turned into what appears to be a port 1433 buffer overflow by exploiting the name resolution vulnerability (used by Slammer) on UDP port 1434 and using the "write the value zero to any writable location" primitive. Such an attack would open enough port 1433 TCP connections to tie up all but one thread of the SQL server, load a bogus stack frame complete with executable payload on one connection, and load fake junk to all of the others. The stacks for these threads could be held in a suspended state by not closing the TCP connections.

Then through UDP port 1434 the attacker would send SQL name resolution exploits that only overwrote the return pointer with its original value but, more importantly, changed the address where the value 0 is written to point to each successive stack. A well placed zero that overwrites the least significant byte of a base pointer on a stack enables linking in a bogus stack frame (this is how the Turkey exploit works). Then by closing the port 1433 TCP connection with the exploit code in it, the stack is unwound until the bogus stack frame hijacks control flow. Because the incoming 0 would not be labeled, and because the row spaces of γ and π would have been mapped from packets for TCP port 1433, it would be easy for DACODA-based analysis to assume that there had been a buffer overflow on port 1433. Fortunately, DACODA records when labeled data is used as addresses so the connection with UDP port 1434 could be identified.

4.3.4 Side Channels

The innd exploit shares something in common with both of the ftp exploits presented here in that the process being attacked does a reverse DNS entry lookup on the IP address of the attacker. It is not clear whether DACODA should include this in the analysis of the attack or not because the attacker could use their DNS name to inject part of the attack into the address space of the vulnerable process but typically will not do so. For all results presented in this paper the DNS traffic is included in the analysis.

Also, parts of the UDP header for Slammer, the source IP address and port, are present in the address space when the bogus control flow transfer occurs. Thus not all of the various parts of an attack can be found in the data fields of TCP and UDP packets; they may come from the packet headers as well.

4.3.5 Encodings and Encryption

The various ASN.1 vulnerabilities found in Microsoft Windows over the past two years [52, bids 9633, 9635, 9743, 10118, 13300], none of which were tested with DACODA, are exposed through several services. They can be exploited through Kerberos on UDP port 88, SSL on TCP port 443, or NTLMv2 authentication on TCP ports 135, 139, or 445.

The malicious exploit and code can be encoded or encrypted with Kerberos, SSL, IPSec, or Base64 encoding (on top of the malicious ASN.1 encoding) [52, bid 9635]. A more advanced exploit for this vulnerability could encrypt most of ϵ, and all of γ and π, and the decryption would be performed by the vulnerable host. The fact that not many vulnerabilities have this property should not be taken to mean that it will be a rare property for zero-day vulnerabilities. Zero-day vulnerabilities will be found in the places that attackers look for them.

Encodings or encryptions of ϵ and γ that are decoded or decrypted by protocol implemented on the machine being attacked are only a challenge for DACODA if the symbolic expressions become too large to handle efficiently or too complex to be useful. In either case DACODA reports this fact, so that a different response than content filtering can be mounted. When symbolic expressions exceed a certain size they become idempotent expressions that denote that a large symbolic expression has been dropped.

4.3.6 Undesirable Predicates

The *wu-ftpd* format string attack helps illustrate what future work is needed before DACODA can consistently analyze real attacks with a high degree of assurance. We used the Hannibal attack from Crandall and Chong [12] where the major portion of the format string is of the form, "`%9f%9f%9f%9f%9f...`". DACODA should, and does, discover predicates for "`%`" and "`f`" but should not discover a strong equality predicate for "`9`" because the format string attack could take the form "`%11f%4f%132f...`".

The *_IO_default_xsputn()* function from glibc sets a variable to 0 and increments that variable for every character printed. When it is done printing the floating point number it subtracts this count from the value 9 calculated by taking the ASCII value 0x39 for "`9`" and subtracting 0x30 (basically, although, as is often the case, reverse engineering by DACODA reveals the actual decoding implementation to be much more convoluted). If this value is equal to zero a strong, explicit equality predicate is discovered by DACODA and the *_IO_default_xsputn()* function moves on (The *__printf_fp()* function does a similar check on the same byte so two predicates are actually discovered). Otherwise the difference is used as the number of trailing zeroes to print and DACODA discovers no predicate. This causes DACODA to discover strong equality predicates for that individual "`9`" if and only if the value on the stack being eaten, which for all practical purposes is random, consumes 9 characters when interpreted as a floating point number without trailing zeroes. The long tokens discovered for the wu-ftpd format string attack in Table 5 are not a good signature but rather represent the fact that a long sequence of data words on the stack require 9 characters to be printed as floating points (including the leading space).

In the Apache exploit the chunked encoding tokens can use any character allowed by the chunked encoding portion of the HTTP protocol, but DACODA discovers predicates because whatever character is used is converted to lower case and compared with a whole array of characters until it is found.

For the innd exploit DACODA generates a token "`test`" as these letters are individually checked against a *d_entry* in the directory containing the various newsgroups on that news server. This token is discovered in the kernel space in the function *d_lookup()*. The "`test`" newsgroup is guaranteed to be there but there is no requirement that the attacker post the original message to this group. The attacker might first log into the news server to request a list of newsgroups on that server and choose a different group every time. Thus the token "`test`" generated by DACODA is not guaranteed to be in every exploit for this vulnerability.

One interesting behavior of the Turkey exploit is that it creates several files or directories with long filenames and then uses these files or directories in some way. This would cause DACODA to discover very long tokens for these filenames (equality between the file name used for creation and the file name used for accessing), except that we added a heuristic that DACODA does not include strong, explicit equality predicates between two labeled symbolic expressions that are both from the attacker.

4.3.7 Lack of a Good Signature for Some Exploits

It is difficult to generalize to a signature for a vulnerability when there is not even a good signature for the exploit. For Slammer the only byte string signature not susceptible to simple polymorphic techniques is the first byte in the UDP packet, "`0x04`". This byte is common to all SQL name resolution requests. The bogus return pointer also has to be a valid register spring and another pointer must point to any writable memory location, but these are not strong predicates. The SQL authentication exploit does not offer much in terms of a signature, either.

The Apache chunk handling exploit, like the wu-ftpd format string exploit, has erroneous predicates in Table 5. This means that all of the tokens with four bytes or more, except the chunked encoding token, are actually not invariant, leaving mostly dots, slashes, dashes, and the new line character, all of which are not uncommon in HTTP traffic. This does not offer a very good invariant signature for content filtering; only the token "`\x0d\nTransfer-Encoding:\x20chunked\x0d\n\x0d\n`" which would block a valid portion of the HTTP protocol. In the ntpd exploit both 4-byte tokens are "`\x00\x00\x00\x00`" which is not uncommon content on any port. The longest token, 8 bytes long, is "`stratum=`" which probably is not uncommon for traffic on UDP port 123.

We did not test any ASN.1 vulnerabilities, but these serve as good examples of just how polymorphic ϵ could be. The ASN.1 library length integer overflow [52, bid 9633] basically has a signature of "`\x04\x84\xFF\xFF\xFF`". The rest of ϵ in this case is identical to any NTLM request over SMB carrying an ASN.1 encoded security token. In fact, the first 445 bytes of all ASN.1 exploits through NTLM [52, bids 9633, 9635, 9743, 10118, 13300] and the Workstation Service [52, bid 9011] exploit are identical. This initial part of the exploit vector is not a good signature unless it is desired that all NTLM requests carrying ASN.1 security tokens be prevented. Furthermore, the Workstation Service results from Table 5 show that the longest string of invariant bytes in this 445-byte sequence is only 23 bytes long. Two other ASN.1 vulnerabilities [52, bids 9635, 13300] have no byte string signature at all to describe them.

As far as purely network-based signature generation methods with no host context, which lack vulnerability information for generalizing observed exploits and predicting future exploits, not a lot of polymorphism is required for a worm not to be detected. Discounting the very long wu-ftpd format string and Turkey tokens which are errors, only one of the 14 exploits has a token of more than 40 bytes. The number 40 is significant since it is the minimum signature size that the first implementation of EarlyBird [37] can discover. A similar result is shown in Section 4.2 of Kim and Karp [21] where the ability to generate signatures falls dramatically when less than 32 bytes of contiguous invariant content are present, which is true for 10 of the 14 exploits in Table 5. Thus EarlyBird and Autograph, in their current implementations, would not be effective against polymorphic versions of between 10 and 13 of the 14 exploits analyzed in this paper.

5. POLY/METAMORPHISM

Based on the results in the previous section, we would now like to formalize polymorphism and metamorphism in ϵ. To be more perspicuous in doing so, and also to guide future work, we describe a model to encompass complexities such as multiple processes, multithreading, and kernel processing of network data by viewing control flow hijacking attacks "from-the-architecture-up." In this way interprocess communication and context switches are viewed simply as physical data transfers in registers and memory. The *Physical Data Requires-Provides model*, or PD-Requires-Provides model, is a requires-provides model [39] for physical data transfers where the focus is on primitives, not vulnerabilities, for reasons that will be discussed in this section.

First we wish to confute the idea that there is a single user-space process that is vulnerable and the attacker opens a TCP connection directly to this process to carry out the exploit. Of the 14 exploits analyzed in Section 4, six involve multiple processes, five involve a significant number of predicates discovered in kernel space, and seven exploit processes that contain multiple threads and are accessible through multiple ports.

The purpose of an exploit is to move the system being attacked from its initial state to a state where control flow hijacking occurs. The series of states the attacker causes the system to traverse from the initial state to control flow hijacking is the attack trace. The attacker causes this traversal of states by sending some set of IP packets that are projected onto the trace of the vulnerable machine as they are interpreted by the protocol implementation on that machine.

The attacker must prevent a satisfactory characterization of the worm traffic by varying bytes in the row spaces of the three projections that do not have a strong equality predicate required of them (polymorphism) or changing the mappings for each infection (metamorphism). In past work [14] we showed that there is a high degree of polymorphism and metamorphism available to the attacker for both γ and π, so we will focus here only on the subject of this paper: ϵ.

5.1 What are Poly- and Metamorphism?

What do we mean when we say polymorphism and metamorphism in ϵ?

5.1.1 *Polymorphism of* ϵ

Some bytes mapped by ϵ by definition are not actually what one might think of when discussing ϵ but should be mentioned for completeness. Filler bytes that serve no other purpose than to take up space, such as the "`XXXXXXX...`" string of bytes in Code Red II, are usually in ϵ but have no strong equality predicate required of them. Usually their placement in ϵ is only because it is required that they are not equal to a NULL terminator or an end of line character, or that they must be printable ASCII characters.

5.1.2 *Metamorphism of* ϵ

We will discuss two kinds of metamorphism: without multithreading and with multithreading. Metamorphism is the more fundamental challenge for DACODA since DACODA is based on symbolic execution of one attack trace and metamorphism in ϵ changes the attack trace.

Without multithreading, there are multiple ways to traverse from the initial state to the control flow hijacking. The three ways of changing this trace are:

1. *Take an equivalent path*: In format string attacks "`%x`" and "`%u`" take different paths but converge so for practical purposes the traces are the same. A couple of examples from the Code Red II exploit are "`.ida`" versus "`.idq`" or the fact that the UNICODE encoding escape sequence "`%u`" can appear anywhere in the GET request between "`?`" and "`=`".

2. *Add paths that are unnecessary*: In the Hannibal attack for the wu-ftpd format string vulnerability the attacker can, after logging in but before carrying out the actual attack, use valid FTP commands that are not useful except that DACODA will discover predicates for them as they are parsed. Pipelining in HTTP 1.1 allows for similar behavior as was pointed out in Vigna et. al. [42].

3. *Changing the order*: In addition to adding paths that are not relevant to the exploit, sometimes paths relevant to the exploit can be arranged in a different order.

What we need to understand metamorphism is a partial ordering on the bytes from the range of ϵ. This partial ordering could help us determine that, for example, the Code Red II buffer overflow is not reachable except through a path in which the token "`%u`" is discovered, and that "`.ida?`" must come before this token and "`=`" must come after. It would also show that "`GET`" must be "`GET`" and not "`GTE`" or "`TEG`". For generating a signature the partial ordering will reveal which tokens are not necessary for control flow hijacking to occur, which tokens can be replaced with other tokens (this will require further analysis such as model checking), and will identify any ordering constraints on those tokens that must occur.

The requires-provides model for control flow hijacking attacks could be as simple as a control flow graph for the whole system. The problem with this is that an attacker with the ability to corrupt arbitrary memory with two threads in the same process can subvert the most basic assumptions (for example, that if a thread sets a local variable to a value it will have the same value until the thread modifies it again). We need a model that can handle multithreading, but first we need to try to understand what a vulnerability-specific signature would need to encompass. To do this we have to discuss what a vulnerability is.

5.2 What is a vulnerability?

What causes a sequence of network packets to be a control flow hijacking attack, the vulnerability, is very subjective.

For buffer overflow exploits it is the fact that a particular field exceeds a certain length; in the case of Slammer it is the length of the UDP packet itself, and for the Turkey exploit the allowable length is exceeded by only one byte. For double *free()* and dangling pointer exploits the exploit is usually caused because a certain token appears twice when it should appear once or is nested such as the constructed bit strings of the ASN.1 dangling pointer vulnerability [52, bid 13300]. Format string write attacks are caused by the presence of a token "`%n`". Integer overflows occur because a particular integer is negative.

One last example puts this problem in perspective. The Code Red II buffer overflow only occurs when at least one "`%u`" token is present which expands all of the ASCII characters to 2 bytes, and the "`u`" character as well as numbers are certainly acceptable in a normal URI. Changing a single bit in the ASCII sequence "`eu1234`" creates "`%u1234`", so the Hamming distance between a valid ASCII GET request of acceptable length and one with a single UNICODE-encoded character that causes a buffer overflow is only one bit! Furthermore, UNICODE encodings in GET requests of normal length are certainly valid within the HTTP protocol or else they would not have been implemented.

There are two ways to create a signature that covers a wide enough set of exploits to be called "vulnerability-specific." One is to add more precision to the signature and use heuristics within the signature generator to look at not only tokens but, for example, also the lengths of fields. Slammer could be stopped by dropping all UDP packets to port 1434 that exceed a certain length. Code Red II could be stopped by dropping all HTTP requests with UNICODE encodings that exceed a certain length. The problem with the Code Red II example is that it requires a lot of parsing of HTTP commands by the network content filter. This is even worse in the case of Scalper because the Apache chunk handling exploit only occurs when a particular integer is negative.

The second way to generate signatures is to relax the requirement that no portion of a valid protocol be dropped. In the Code Red II example above, we could simply drop all HTTP requests with UNICODE encodings since normal HTTP traffic typically will not use them. For Scalper we could drop all HTTP traffic with the token "`\x0d\nTransfer-Encoding:\x20chunked\x0d\n\x0d\n`" which will not allow any chunked encodings, and is in fact the rule that Snort [53] uses. In other words, it may be acceptable to block a valid portion of a protocol (or even an entire protocol by blocking its ports) if that portion is not often used by legitimate traffic. Most vulnerabilities are discovered in code that is not frequently used. These coarse responses may sometimes be the most effective, but the challenge is knowing, upon capturing an exploit for an unknown vulnerability, that the protocol involved or the specific part of it where the vulnerability lies is rarely used, something that would need to be profiled over a long period of time.

The first of these alternatives leaves us "on the horns of a dilemma" [49] in terms of false positives and false negatives without a detailed semantic understanding of how the exploit works. It also is not amenable to byte string signatures, even those based on small sets of tokens, so something more semantically rich will have to be devised. This is the challenge that we hope to address in future work.

The second alternative will create a great number of false positives if the worm exploits a vulnerability that is in a part of a protocol that is used often. *This is because nearly all of the tokens in Table 5 are protocol framing and not related to the actual vulnerability.* Buffer overflows have been found in Microsoft libraries for both JPEG parsing [50, MS04-028] and JPEG rendering [50, MS05-038]. If a worm exploited such vulnerabilities, it would create many false positives if the worm content filtering mechanism blocked all HTTP responses containing JPEGs.

5.3 The PD-Requires-Provides Model

Metamorphic techniques that use arbitrary memory corruption primitives in multithreaded applications to build complex exploits require a model that views the system from the same perspective as the attacker will: the raw machine. This "from-the-architecture-up" view of the system will allow us to abstract away system details that lead to assumptions that the attacker can invalidate. This is the motivation behind the PD-Requires-Provides model.

5.3.1 Requirements and What They Provide

An attacker can only cause a state transition along the attack trace through the execution of a machine instruction that uses data from the range of ϵ. We will assume a Pentium processor and a sequential consistency memory model. Although the Pentium uses a processor consistency model and multiprocessing is becoming more and more common, it may be too pessimistic at this time to assume that the attacker could exploit a race condition between the memory and the write buffers of two high speed processors. It should be noted, however, that race conditions between threads have been demonstrated to permit remote code execution [47].

Treating each machine instruction that is executed as an atomic event, we can say that to *provide* a side effect needed by the exploit there is something *required* of the inputs. A side effect the attacker would like to provide could be a write to memory, a write to a register, a write to a control flag, or a branch predicated on a control flag. It could be required that an input to the instruction be a certain value from the range of ϵ, that the address used to load an input be from the range of ϵ, or that a control flag have been predicated on a comparison of data from the range of ϵ (providing a write to the program counter EIP).

5.3.2 Slammer Example

Suppose we want to exploit the vulnerability used by Slammer to write the value 0 to the virtual address 0x0102aabb in the SQL server process. It is required that we get the value 0x0102aabb into the EAX register before the instruction "`MOV [EAX],0`" is executed. This requires that we send a long UDP packet to port 1434. Specifically, when the Ethernet packet is received it is required that the "`IN DX`" instructions that read the packet read a carefully crafted UDP packet two bytes at a time to provide that the packet be stored in a buffer and interpreted by the Windows kernel in a certain way. When the Windows kernel checks the 24th byte of the packet it is required that this memory location hold the value 0x11 so that when it is loaded into a register and compared to 0x11 the branch will be taken where the kernel interprets it as a UDP packet. Similar requirements on the port number and destination address will provide the state transitions of the kernel recognizing a packet for the SQL server process and then context switching into that process providing us with the ability to read and write the physical memory of that process.

The SQL thread chosen to handle the request will then context switch to the kernel and back twice to obtain the source address and port number information and then to read the packet into its own memory space. Then it is required of each byte that it not be equal to "`0x00`" or "`0xFF`" in order to reach the buffer overflow condition. It is also required of the first data byte of the UDP packet to be equal to "`0x04`" so that the vulnerable function is reached through the sequence "`MOV EDX,[EBP+fffffbf4]; JMP [42cfa23b+EDX<<2]`". Then before "`MOV [EAX],0`" the EAX register must hold the attacker's desired arbitrary address (0x0102aabb), provided by the instruction, "`MOV EAX, [EBP+10]`" which requires the value 0x0102aabb to be at "`[EBP+10]`". Finally, all of this will provide the primitive that the value 0 is written to the virtual address 0x0102aabb of the SQL server process which may be required for some exploit such as the one suggested in Subsection 4.3.3.

5.3.3 Should Focus on Primitives, not Vulnerabilities

The goal of a signature generation algorithm based on DACODA, then, should be to, given the partial ordering constructed for a single exploit as analyzed by DACODA, identify the primitive most valuable to the attacker in generating new exploits and generate a signature that prevents that primitive. This will most likely have to be done with heuristics. A good heuristic is that arbitrary write primitives are valuable to an attacker, which will be revealed by a write provided by a requirement that the address used for the write was data from the range of ϵ. That requirement was provided by some other requirement, which in turn was provided by another requirement, giving us a way to work backwards and generate a primitive-specific signature from the partial ordering. Another good heuristic is that saved base pointers and return pointers on the stack should not be overwritten by long fields, but this requires knowing which field is too long which in turn requires knowing what the delimiters between fields are for that particular protocol (information that will have to be extracted from the partial ordering). Similar heuristics could be made for any sort of primitive that an attacker might find valuable in building exploits. The point is that an attacker who searches for a zero-day vulnerability is not so much searching for a vulnerability as for a useful primitive for generating exploits.

6. FUTURE WORK

DACODA can be useful toward a variety of objectives, several of which we will now discuss. In this paper we have used DACODA to analyze known exploits as a quantitative, empirical analysis of the amount of polymorphism available to an attacker within the exploit vector. DACODA may also be used as a honeypot technology to perform the same analysis on zero-day worms exploiting unknown vulnerabilities for signature generation. This same idea was employed in Vigilante [10] and suggested as future work for Polygraph [28] and TaintCheck [29].

Other possible future work for DACODA is to use predicates discovered by DACODA and heuristics about different memory corruption errors to narrow the search space of a random "fuzz tester" [26, 27]. It would be possible to find buffer overflows and other remote vulnerabilities in both user-space and the kernel this way. This system would be similar to two recent papers on automatically generating test cases [4, 18] but would operate on a full system without the source code and find remote vulnerabilities.

Full system symbolic execution has many other security applications, but it was pointed out in Cohen's seminal paper on computer viruses [9] that the general problem of precisely marking information flow within a system was shown to be NP-complete by Fenton [17]. DACODA is able to analyze the exploit vector part of an attack because the code being executed is code chosen by the owner of the host such as the operating system and software she chooses to install. After control flow is hijacked the computational complexity of information flow tracking is more than a theoretic problem because the attacker can use techniques such as phi-hiding to obfuscate information flow in a cryptographically strong manner [45].

7. CONCLUSION

This paper presented DACODA and provided a quantitative look at the exploit vectors mapped by ϵ for 14 real exploits. These results and our experiences with DACODA discussed in this paper offer practical experience and sound theory towards reliable, automatic, host-based worm signature generation. We have shown that 1) single contiguous byte string signatures are not effective for content filtering, and token-based byte string signatures composed of smaller substrings are only semantically rich enough to be effective for content filtering if the vulnerability lies in a part of a protocol that is not commonly used, and that 2) whole-system analysis is critical in understanding exploits. As a consequence we conclude that the focus of a signature generation algorithm based on DACODA should be on primitives rather than vulnerabilities.

8. ACKNOWLEDGMENTS

This work was supported by NSF ITR grants CCR-0113418 and ACI-0220147. We are also very grateful to our shepherd, Dan Boneh, and many people who discussed the Minos and DACODA projects with us or read earlier versions of this paper, including Daniela Alvim Seabra de Oliveira, Timothy Sherwood, Helen Wang, and everyone in the U.C. Davis malware reading group and security lab seminar. The anonymous reviewers also provided very insightful comments. We would also like to thank the Bochs developers.

9. REFERENCES

[1] P. Akritidis, E. P. Markatos, M. Polychronakis, and K. Anagnostakis. Stride: Polymorphic sled detection through instruction sequence analysis. In *20th IFIP International Information Security Conference*.

[2] Barnaby Jack. Remote Windows Kernel Exploitation-Step Into the Ring 0.

[3] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM conference on Computer and communication security*, pages 281–289. ACM Press, 2003.

[4] C. Cadar and D. Engler. Execution generated test cases: how to make systems code crash itself. In *SPIN*, 2005.

[5] S. Chen, J. Xu, and E. C. Sezer. Non-control-hijacking attacks are realistic threats. In *USENIX Security Symposium 2005*, 2005.

[6] R. Chinchani and E. van den Berg. A fast static analysis approach to detect exploit code inside network flows. In *RAID*, 2005.

[7] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns, 2003.

[8] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (Oakland 2005)*, Oakland, CA, USA, May 2005.

[9] F. Cohen. Computer viruses: theory and experiments. In *7th DoD/NBS Computer Security Conference Proceedings*, pages 240–263, September 1984.

[10] M. Costa, J. Crowcroft, M. Castro, and A. Rowstron. Can we contain internet worms? In *HotNets III*.

[11] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of Internet worms. In *SOSP '05: Proceedings of the twentieth ACM Symposium on Operating Systems Principles*, New York, NY, USA, 2005. ACM Press.

[12] J. R. Crandall and F. T. Chong. A Security Assessment of the Minos Architecture. In *Workshop on Architectural Support for Security and Anti-Virus*, Oct. 2004.

[13] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, December 2004.

[14] J. R. Crandall, S. F. Wu, and F. T. Chong. Experiences using Minos as a tool for capturing and analyzing novel worms for unknown vulnerabilities. In *Proceedings of GI SIG SIDAR Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2005.

[15] H. Dreger, C. Kreibich, V. Paxson, and R. Sommer. Enhancing the accuracy of network-based intrusion detection with host-based context. In *Proceedings of GI SIG SIDAR Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA), 2005.*

[16] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, 2002.

[17] J. Fenton. Information protection systems. In *Ph.D. Thesis, University of Cambridge*, 1973.

[18] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI*, 2005.

[19] S.-S. Hong, F. Wong, S. F. Wu, B. Lilja, T. Y. Jansson, H. Johnson, and A. Nelsson. TCPtransform: Property-oriented TCP traffic transformation. In *Proceedings of GI SIG SIDAR Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA), 2005.*

[20] S.-S. Hong and S. F. Wu. On interactive Internet traffic replay. In *RAID*, 2005.

[21] H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *USENIX Security Symposium*, pages 271–286, 2004.

[22] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[23] O. Kolesnikov and W. Lee. Advanced polymorphic worms: Evading IDS by blending in with normal traffic.

[24] C. Kreibich and J. Crowcroft. Honeycomb: creating intrusion detection signatures using honeypots. *SIGCOMM Comput. Commun. Rev.*, 34(1):51–56, 2004.

[25] C. Krügel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *RAID*, 2005.

[26] B. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report, 1995.

[27] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the Association for Computing Machinery*, 33(12):32–44, 1990.

[28] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the IEEE Symposium on Security and Privacy, May*, 2005.

[29] J. Newsome and D. Song. 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 (NDSS 05)*, Feb. 2005.

[30] A. Pasupulati, J. Coit, K. Levitt, S. Wu, S. Li, R. Kuo, and K. Fan. Buttercup: On network-based detection of polymorphic buffer overflow vulnerabilities. In *9th IEEE/IFIP Network Operation and Management Symposium (NOMS'2004)*, 2004.

[31] U. Payer, P. Teufl, and M. Lamberger. Hybrid engine for polymorphic shellcode detection. In *Proceedings of GI SIG SIDAR Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA), 2005.*

[32] T. H. Ptacek and T. N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc., Suite 330, 1201 5th Street S.W, Calgary, Alberta, Canada, T2R-0Y6, 1998.

[33] C. Raiu. Holding the Bady. In *Virus Bulletin*, 2001.

[34] S. Rubin, S. Jha, and B. P. Miller. Automatic generation and analysis of NIDS attacks. In *20th Annual Computer Security Applications Conference (ACSAC)*.

[35] S. Rubin, S. Jha, and B. P. Miller. Language-based generation and evaluation of NIDS signatures. In *IEEE Symposium on Security and Privacy, Oakland, California, May*, 2005.

[36] M. E. Russinovich and D. A. Solomon. *Microsoft Windows Internals, Fourth Edition*. 2004.

[37] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *OSDI*, 2004.

[38] P. Szor. *The Art of Computer Virus Research and Defense*. 2005.

[39] S. J. Templeton and K. Levitt. A requires/provides model for computer attacks. In *NSPW '00: Proceedings of the 2000 workshop on New security paradigms*, pages 31–38, New York, NY, USA, 2000. ACM Press.

[40] T. Toth and C. Krügel. Accurate buffer overflow detection via abstract payload execution. In *RAID*, pages 274–291, 2002.

[41] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. Rifle: An architectural framework for user-centric information-flow security. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, December 2004.

[42] G. Vigna, W. Robertson, and D. Balzarotti. Testing Network-based Intrusion Detection Signatures Using Mutant Exploits. In *Proceedings of the ACM Conference on Computer and Communication Security (ACM CCS)*, pages 21–30, Washington, DC, October 2004.

[43] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. 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*, pages 193–204. ACM Press, 2004.

[44] V. Yegneswaran, J. T. Giffin, P. Barford, and S. Jha. An architecture for generating semantics-aware signatures. In *USENIX Security Symposium*, 2005.

[45] A. Young and M. Yung. *Malicious Cryptography: Exposing Cryptovirology*. 2004.

[46] bochs: the Open Source IA-32 Emulation Project (Home Page), http://bochs.sourceforge.net.

[47] eEye advisory for the DCOM RPC Race Condition (http://www.eeye.com/html/research/advisories/AD20040413B.html).

[48] eEye advisory for the LSASS buffer overflow (http://www.eeye.com/html/research/advisories/AD20040413C.html).

[49] General William T. Sherman, as quoted in B. H. Liddell Hart, *Strategy*, second revised edition.

[50] Microsoft advisory MSXX-YYY (http://www.microsoft.com/technet/security/bulletin/MSXX-YYY.mspx).

[51] QEMU (Home Page), http://fabrice.bellard.free.fr/qemu/.

[52] Security Focus Vulnerability Notes, (http://www.securityfocus.com), bid == Bugtraq ID.

[53] SNORT: The open source network intrusion detection system (http://www.snort.org). 2002.