Analyzing Network Traffic To Detect Self-Decrypting Exploit Code [*]

Qinghua Zhang, Douglas S. Reeves, Peng Ning, S. Purushothaman Iyer
Cyber Defense Laboratory, Computer Science Department
North Carolina State University, Raleigh, NC 27695-8207
{qzhang2, reeves, pning, purush}@ncsu.edu

ABSTRACT

Remotely-launched software exploits are a common way for attackers to intrude into vulnerable computer systems. As detection techniques improve, remote exploitation techniques are also evolving. Recent techniques for evasion of exploit detection include *polymorphism* (code encryption) and *metamorphism* (code obfuscation). This paper addresses the problem of detecting in network traffic polymorphic remote exploits that are encrypted, and that self-decrypt before launching the intrusion. Such exploits pose a great challenge to existing malware detection techniques, partly due to the non-obvious starting location of the exploit code in the network payload.

We describe a new method for detecting self-decrypting exploit codes. This method scans network traffic for the presence of a decryption routine, which is characteristic of such exploits. The proposed method uses static analysis and emulated instruction execution techniques. This improves the accuracy of determining the starting location and instructions of the decryption routine, even if self-modifying code is used. The method outperforms approaches that have been previously proposed, both in terms of detection capabilities, and in detection accuracy.

The proposed method has been implemented and tested on current polymorphic exploits, including ones generated by state-of-the-art polymorphic engines. All exploits have been detected (i.e., a 100% detection rate), including those for which the decryption routine is dynamically coded, or self-modifying. The false positive rate is close to 0%. Running time is approximately linear in the size of the network payload being analyzed.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Network**]: General—*Security and protection*; D.3.3 [**Programming languages**]: Language Constructs and Features—*Polymorphism*

[*]This work is supported by the National Science Foundation (NSF) under grant CNS-0627505.

General Terms

Security

Keywords

Polymorphic, Exploit Code, Decryption, Detection, Static Analysis, Emulation

1. INTRODUCTION

In recent years, malicious code has become an omnipresent and dangerous threat to the Internet infrastructure. The mono-culture property of the hardware and software makes it possible to explore a single vulnerability to compromise a large number of hosts. A typical example are the Internet worms which are self-propagating and self-replicating. Worms frequently use *remote exploit code* to intrude into a vulnerable system attached to the Internet, and then compromise other vulnerable systems attached to the Internet. The automatic nature of worms makes them virulent and destructive. According to *Computer Economics* [2], the estimated worldwide damages caused by three well-known worms, which are *Code Red*, *Nimda* and *Slammer* worms in year 2001, exceeded $4 billion. Detection of exploit code in network traffic is a crucial task in stopping the spread of worms.

Signature-based intrusion detection is the most popular approach for detecting remote exploits that target vulnerable hosts. Some well-known and very successful systems include Snort [8] and Bro [7]. While signature-based intrusion detection has been very successful, these detection systems can be defeated by exploits that use advanced concealment techniques. Two important recent such techniques are *polymorphism*, which makes use of exploit code encryption, and *metamorphism*, which uses a variety of code obfuscation techniques to confound static signature-checking.

Recently, there has been substantial research on defeating such concealment methods. Approaches that use static analysis [23, 22, 10, 9] have shown promise in detecting a specific and non-trivial class of polymorphic exploits. These exploits target buffer overflow vulnerabilities, accounting for more than 20% of the vulnerabilities reported by CVE [1]. This class of exploits contains program-like code that has distinctive control flow and data flow characteristics that can be detected. For instance, figure 1 shows a characteristic structure of polymorphic exploits, produced by one of the available exploit toolkits [5, 6, 4].

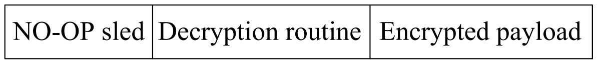


Figure 1: A typical structure of polymorphic exploit codes on buffer overflow vulnerabilities

The effectiveness of such static analysis approaches depends on how well the program-like payload (e.g. the decryption routines) can be distinguished from a) non-code data and from b) non-exploit code. There are several significant challenges. First, exploit code is often hidden inside network traffic at a non-obvious starting location , and interspersed with data (whether valid or not) [10]. Code bytes cannot be distinguished from data bytes solely by the use of disassembly, due to the compactness of Intel instruction set. Second, exploit code that is "visible" (i.e., unencrypted) is usually manually crafted, and does not follow the same conventions as executable programs generated by a compiler. For instance, hand-crafted exploit code may use overlapping instructions, or self-modifying instructions, expressly for the purpose of defeating static disassembly.

Available static-analysis approaches [9, 10, 22, 23] have only addressed the above challenges to a limited degree. They are not robust against exploits which employ static analysis-resistant techniques such as self-modifying and indirect control transfer instructions [19], partly due to the non-obvious starting location of the exploit code. The methods proposed by Toth and Kruegel [22] and Akritidis *et al.* [9] focused on NO-OP sleds which may be missing in advanced exploit code [12, 18]. The method used by Chinchani *et al.* [10] to extract the control flow of the exploits is not resistant to data injection attack. Wang *et al.* [23] proposed a code abstraction method to distill useful instructions from an instruction sequence to detect exploits. However the code abstraction is based on a data-flow anomaly rule that an instruction referencing an undefined variable is redeemed useless. This is easily evaded by some obfuscation techniques. For instance, two instructions referencing the same undefined variable can still be useful, if they can be used to clear the registers for initialization. If an attacker exploits this property, most of their useful instructions would probably not be detected by a chaining effect. In addition, none of these approaches offer a mechanism to clearly identify the starting location of the polymorphic exploit code in network traffic.

Polychronakis *et al.* [19] proposed a method that uses instruction emulation to more effectively identify self-modifying polymorphic exploit code than is possible with static analysis. Their approach does not, however, provide a comprehensive mechanism to identify the starting location of polymorphic exploit code in network traffic. The method will be very slow if all potential starting locations are tried, but simple heuristics for narrowing down the possible starts of exploit code may miss some attacks.

In this paper we present a new method to detect self-decrypting exploit code. The proposed method harnesses static analysis and emulated instruction execution to find the starting location and identify the instructions of the polymorphic exploit code. Previous approaches [19, 10, 23, 22, 9] do not offer such capabilities. In addition, the proposed method can detect polymorphic exploit code that is self-modifying, which static analysis has previously been unable to detect.

The proposed method works by scanning the network traffic for the presence of a decryption routine, which is characteristic of such exploits. First, it identifies the possible starting locations of a decryption routine by looking for a form of *GetPC* code. This is a generally inevitable component of the decryption routine, which is used to find the absolute address of the encrypted exploit code. Second, it finds the actual decryption instructions by a novel two-way traversal of the code, as well as by using standard backward data flow analysis. Third, it identifies self-modifying decryption routines through emulated execution of already found decryption instructions. Last, it verifies the detected code is a decryption routine by checking whether it satisfies two properties that are typical of such code.

The proposed method has been implemented and evaluated against real polymorphic exploits produced by Metasploit [4], and also those produced by polymorphic engines [5, 6]. It achieves a 100% detection rate on the polymorphic exploits which use statically coded decryption routines. It likewise has a 100% detection rate for decryption routines that are self-modifying . The method was also tested on typical network traffic not containing polymorphic exploits, and on Windows executable files. The false positive rate is approximately .0002% and .01% for these two categories, respectively. We also measure the running time of our non-optimized implementation. The running time is roughly linear in the size of the traffic being analyzed, and is between 1 and 2 MB/s.

The rest of this paper is organized as follows. Section 2 describes related work. Section 3 presents our method. Section 4 demonstrates our evaluation. Section 5 gives the attack analysis. Section 6 concludes the paper.

2. RELATED WORK

The two major approaches to detection of polymorphic exploit code are static-analysis [10, 9, 22, 23] and emulation [19]. These have been briefly described above. Two other approaches for detection of polymorphic exploits in network traffic are signature-based methods, and data mining methods. These are now briefly described.

Polygraph [16] and Hamsa [14] are examples of the signature-based approach. These methods generate polymorphic worm signatures by finding common invariant content substrings among multiple polymorphic worm samples. This approach requires a network traffic classifier to preselect suspicious traffic for training. The detection and false positive rates depend on the effectiveness of the classifier. If the background "noise" (network traffic not containing exploits, but selected by the classifier) is significant, the accuracy can be significantly reduced. Newsome *et al.* [17] presented an attack that causes the signature learning approach to fail. Nemean [25] is a method that uses protocol semantics to group similar worm traffic, and that uses machine-learning techniques to generate connection and session level signatures. This approach requires detailed protocol specification for every application protocol. It is also sensitive to background noise. In general, this category of methods has a high maintenance cost, in the sense that signature repositories need to be constantly updated, as new polymorphic variants are encountered.

The second category uses data-mining [18]. This approach combines neural networks with a simple NO-OP sled detector (as used in [22]) to detect exploit code. The neural network has to be carefully trained with negative and positive data sets, which highly affects its detection rates. In practice, high quality training sets may be difficult to obtain and keep updated.

There is considerable work on malware detection or prevention at the host level. These methods provide insight into the form of malware, and the run time environment. A recent method named PolyUnpack [20] automatically identifies and extracts the hidden-code bodies of unpacking-executing malware, with knowledge of the instance's static code model. Christodorescu et al [11] proposed to detect malware through the use of semantic behavior models, called templates. One of the presented templates models is the decryption loop of polymorphic malware. However, this method does not provide a way to model self-modifying decryption loops. How to define a general semantic behavior model remains an open problem. Sidiroglou *et al.* [21] proposed an end-point architecture to automatically repair software flaws to counter various attacks. This approach requires information about the source code.

Work on static binary disassembly is also relevant to this work. There are two widely used disassembly techniques. The *linear sweep* method, which decodes bytes sequentially, has difficulties distinguishing between embedded data and actual instructions. Therefore it can be defeated by data injection attacks, and by other attacks, such as the use of overlapping instructions. Figure 2(a) shows an example where this is the case. *Recursive traversal*, which decodes bytes by following the control flow of the program, can better deal with such attacks. However, it requires the entry point of the program to be known in advance. Moreover, the target address of a branch instruction cannot always be statically determined by recursive traversal. In this case, linear sweep may recover more valid instructions. Kruegel *et al.* [13] proposed an advanced disassembly technique. This method used the program's control flow graph and statistical techniques to correctly identify a large fraction of the program's instructions. However, the assumptions of this method are not fully compatible with the requirements for disassembling code in network traffic, i.e the absent information of program starting location. In addition, it does not correctly handle code in which self-modifying and/or overlapping instructions are used.

3. THE PROPOSED METHOD

This section describes a network-level *hybrid* method for the detection of polymorphic exploit code. First, the detection methods based on static binary code analysis [10, 23, 22, 9] can possibly identify the control flow and data flow information of the exploit code. However they cannot well handle static analysis resistant (i.e. self-modifying) polymorphic exploit code which will not reveal its actual form until it is actually executed. Second, the detection method based on emulated execution of network traffic [19] is doing better in this aspect. However it will incur a high processing overhead if each instruction sequence is executed. Simple strategies to select possible instruction sequences for execution will miss some attacks.

We are motivated to explore whether it is possible to detect such highly obfuscated polymorphic exploit code by combining these two types of techniques.

We now first give an overview to the proposed method.

3.1 Overview of the Proposed Method

The overall idea is to scan the network traffic for the presence of the decryption routine which is characteristic of polymorphic exploit code. Static analysis is used to locate the decryption routine inside the network traffic. Limited emulation of instruction execution is performed to reveal concealed components such as self-modifying instructions of the decryption routine. Moreover, a heuristic approach is explored to further increase the overall accuracy.

More specifically, a form of *GetPC* code is first looked for as the basic means of locating the start of the decryption routine. *GetPC* code is a generally inevitable component of a decryption routine. As argued by Polychronakis *et al.* [19], reliable exploit code should avoid any hard-coded absolute addressing. Therefore, the decryption routine must have some way to dynamically determine the address of the encrypted payload in the vulnerable program's address space, in order to modify it. This is accomplished by *GetPC* code, which computes absolute addresses as offsets from the current value of the program counter (the *PC*). The *GetPC* code should be among the very few instructions that cannot be self-modifying or concealed, and it also should be among the very few first functional instructions of a decryption routine. Therefore, detection of *GetPC* code helps localize the *start* of the decryption routine.

Then, the rest of the decryption routine is found by traversing the bytes from where the *GetPC* code is found, and looking for a loop in the control flow structure. Loops are likely to occur in decryption routines for the simple reason that decryption of a sequence of bytes is a very repetitive process. Recursive traversal disassembly is generally useful enough to derive the control flow structure if no complex static analysis resistant techniques are involved. Otherwise, the task is accomplished by a novel two-way traversal and backward dataflow analysis to more precisely pinpoint the non-concealed instructions of the decryption routine, and by emulated instruction execution on already found instructions to reveal the concealed component, i.e. the self-modifying part of the decryption routine.

Finally, the detected code is verified to improve the overall accuracy. The proposed approach checks whether the detected code satisfies two properties that we observe from typical decryption routines. These properties have not been used for this purpose previously.

We now describe the method in detail, starting with the decryption routine localization.

3.2 Decryption Routine Localization

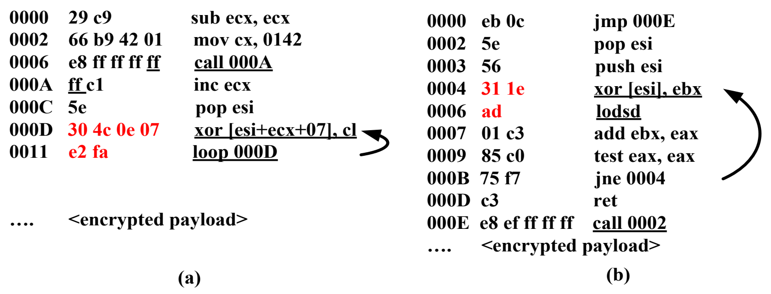
Generally speaking, a decryption routine is suspected if the control flow structure shows the existence of a loop.

3.2.1 General Approach

Our general approach works by first finding the starting point of the decryption routine and then using recursive traversal to find the loop structure of the decryption routine.

Starting Point Localization
The first step is to find the starting instruction of the decryption routine which is hidden within the network traffic. This is done by scanning network traffic for candidate *seeding instructions* of *GetPC* code. We now explain *GetPC* code and seeding instructions.

GetPC code is a generally inevitable component of the



```
0000   29 c9            sub ecx, ecx
0002   66 b9 42 01      mov cx, 0142
0006   e8 ff ff ff ff   call 000A
000A   ff c1            inc ecx
000C   5e               pop esi
000D   30 4c 0e 07      xor [esi+ecx+07], cl
0011   e2 fa            loop 000D

....   <encrypted payload>

                 (a)
```

```
0000  eb 0c           jmp 000E
0002  5e              pop esi
0003  56              push esi
0004  31 1e           xor [esi], ebx
0006  ad              lodsd
0007  01 c3           add ebx, eax
0009  85 c0           test eax, eax
000B  75 f7           jne 0004
000D  c3              ret
000E  e8 ef ff ff ff  call 0002
....  <encrypted payload>

                (b)
```

Figure 2: Disassembly of decryption routines for a) Countdown b) JmpCallAdditive encoders. In each figure, the left-most column shows instructions' addresses represented in `hex` format; the middle column shows the actual instruction bytes; the rightmost column shows the decoded instructions. The underlined instructions are the seeding instruction, the instruction for decrypting the encrypted exploit payload and the instruction for updating the address of encrypted exploit payload. For both examples, a loop structure is presented. In figure a), instruction `call 000A` at address `0006` and `pop esi` at address `000C` are the *GetPC* code of this example. Similarly, in figure b), instruction `call 0002` at address `000E` and `pop esi` at address `0002` are the *GetPC* code.

decryption routine. It is used to dynamically determine the address of the encrypted payload in the vulnerable program's address space, in order to modify it. Polychronakis *et al.* [19] identified two [1] feasible and easy forms of *GetPC* code. One way is through a `call` instruction. Execution of a `call` instruction pushes the return address (the PC) onto the stack. The decryption routine when executed can easily read this return address from the stack. The second way is through a `fnstenv` instruction, which stores the current *FPU* environment that includes the program counter of a preceding *FPU* instruction, in an area of memory specified by the instruction. Then the value of this program counter can be read by a following instruction and used to compute the absolute address of the encrypted payload. Examples of *GetPC* code are shown in figure 2.

We term the `call` or `fnstenv` *seeding instructions*. A seeding instruction stores a program counter (*PC*) of one decryption routine instruction as the base address for later instructions to compute absolute addresses of the encrypted payload as offsets from the base address. Seeding instructions are the key instructions for *GetPC* code to work. The number of candidate *seeding instructions* in the Intel instruction set is expected to be limited.

By scanning the network packet for the seeding instruction (`call`, `fnstenv` etc.) of *GetPC* code, the start of a decryption routine can be *coarsely* located. Each such instruction found is treated as if it belongs to a decryption routine.

Recursive Traversal To Detect Decryption Loop Structure.
Recursive traversal is a standard disassembly technology. It is robust against data-injection attacks, in which code is interleaved with data. The proposed method uses it to find the control flow structure of the decryption routine. Once a loop is detected during recursive traversal, this is a candidate for a decryption routine. However, a recursive traversal may be hindered by indirect addressing branch instructions, and the loop structure can be hidden by self-modification techniques. An enhanced approach can address these two issues. The approach uses (a) two-way traversal and backward data-flow analysis, and (b) a limited emulation of instruction execution.

3.2.2 Enhanced Approach

The enhanced approach deals with the two issues when a self-modifying decryption routine is used and the indirect addressing branch instruction is used.

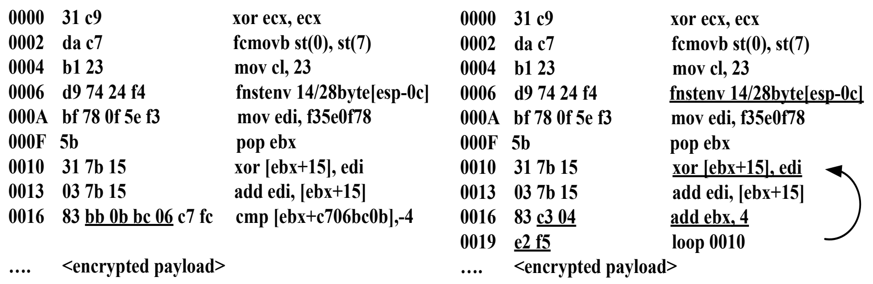
Two-way Traversal and Backward Data Flow Analysis To Find Decryption Instructions.
The enhanced method uses both forward *and* backward traversal of bytes from the seeding instruction to find the rest instructions of the decryption routine. Forward traversal, as usual, recursively decodes the bytes by following the control flow, starting at the seeding instruction. It can find the instructions that are data-flow dependent on the *GetPC* code. This includes the instructions directly responsible for data decryption. Backward traversal decodes bytes in a reverse direction of the control flow, also starting at the seeding instruction. Backward traversal is needed since the seeding instruction *may not* be the very first instruction of the decryption routine, i.e. the initialization instructions. This analysis step is quick if the seeding instruction is close to the start of the decryption routine. Multiple instruction sequences could be found during a backward traversal due to the self-synchronization property of the Intel instruction set [2]. The enhanced method uses backward data flow analysis to determine whether a backward traversal is demanded and which instruction sequence found during such a traversal actually belongs to the decryption routine that exists before the *GetPC* code.

First, the method performs forward traversal which starts at the seeding instruction and follows the control flow to dissemble the byte sequences.

Then, the method triggers a backward data flow analysis if a *target instruction*, an instruction that is either (a) an instruction that writes to memory, or (b) a branch instruction with indirect addressing, is encountered during the forward traversal. When the target instruction is (a), it could be an instruction used for decrypting the hidden loop or the encrypted payload. When the target instruction is (b), it could

[1] M. Polychronakis *et al.* [19] also mentioned a third form of *GetPC* code which is to exploit the structure exception handling(SEH) mechanism of Windows. However they mentioned this technique is not feasible with advanced version of Windows.

[2] A set of bytes is self-synchronizing if it disassembles to the same instruction sequence even if slightly different starting points are chosen. Please refer [10] for more detail.



```
0000  31 c9                xor ecx, ecx
0002  da c7                fcmovb st(0), st(7)
0004  b1 23                mov cl, 23
0006  d9 74 24 f4          fnstenv 14/28byte[esp-0c]
000A  bf 78 0f 5e f3       mov edi, f35e0f78
000F  5b                   pop ebx
0010  31 7b 15             xor [ebx+15], edi
0013  03 7b 15             add edi, [ebx+15]
0016  83 bb 0b bc 06 c7 fc cmp [ebx+c706bc0b],-4
....  <encrypted payload>
```

(a)

```
0000  31 c9                xor ecx, ecx
0002  da c7                fcmovb st(0), st(7)
0004  b1 23                mov cl, 23
0006  d9 74 24 f4          fnstenv 14/28byte[esp-0c]
000A  bf 78 0f 5e f3       mov edi, f35e0f78
000F  5b                   pop ebx
0010  31 7b 15             xor [ebx+15], edi
0013  03 7b 15             add edi, [ebx+15]
0016  83 c3 04             add ebx, 4
0019  e2 f5                loop 0010
....  <encrypted payload>
```

(b)

Figure 3: Disassembly of Self-Modifying decryption routine for ShikataGaNai encoder. a) Before Execution b) After Execution. In each figure, the leftmost column shows instructions' addresses represented in `hex` format; the middle column shows the actual instruction bytes; the rightmost column shows the decoded instructions. The `fnstenv` instruction is the seeding instruction. The `xor [ebx+15], edi]` is the instruction for decrypting the self-modifying decryption routine and encrypted exploit payload. Instruction `fcmovb st(0), st(7)` at address `0002`, `fnstenv 14/28byte[esp-0c]` at address `0006` and `pop ebx` at address `000F` are the *GetPC* code of this example. The loop structure is revealed after execution.

be an instruction to obfuscate the control flow. Either of the instructions is significant. Backwards data-flow analysis is a popular technique for program analysis [15]. Here we use it to find instructions on which the target instruction has data-flow dependency (i.e. we follow backwards the $define-use$ chain) in order to determine the operands (i.e. write-to address and write-to value) of the target instruction. If all the required variables have been defined till the seeding instruction, then there is no non-*GetPC* decryption routine code that exists earlier than the seeding instruction. Otherwise, there must be.

Finally, the method performs backward traversal on demand, determined as described above. Backward traversal decodes bytes in a reverse direction of the control flow. It is implemented using *breadth first search*, starting at the seeding instruction. First the set of instructions that directly reach the seeding instruction at byte offset i of the input network traffic are found. This set will possibly contain branch instructions whose target is the instruction at offset i. The set may also contain non-branching instructions immediately preceding the seeding instruction. Then instructions reaching instructions in this set are found, etc. A branch instruction using indirect addressing is unlikely to appear prior to the seeding instruction in the control flow for the simple reason that the *GetPC* code must be executed first. The same is true for self-modifying instructions.

It must be decided which instruction sequence found during the backward traversal actually belongs to the decryption routine and exists before the *GetPC* code. Backwards data-flow analysis is used again. To choose which instruction sequence contains these code, we pick one that defines all the rest variables or is the longest of multiple qualified instruction sequences.

Figure 3(a) is an example to illustrate this two-way traversal and backwards data-flow analysis. First, forward traversal is performed to decode the byte sequences, starting at address `0006` where the seeding instruction `fnstenv 14/28 byte[esp-0c]` is found. The process is continued till instruction `xor [ebx + 15], edi` at address `0010`, a target instruction, is encountered. Then a round of backward data flow analysis is triggered to find previous instructions that determine the operands of this target instruction. `ebx` and `edi` are variables of the operands of this target instruction. Their values are defined by instruction `pop ebx` at address `000F` and `mov edi, f35e0f78` at address `000A`. Instruction `pop ebx` reads a 4-byte value from the stack referenced through register `esp` by default. The left task is to find the actual value stored at `[esp]` that will be used to define or assigned to `ebx` through `pop ebx`. According to the semantics of `fnstenv`, it will store the program counter (PC) of a FPU instruction preceding it at *twelve* bytes from the address specified. Hence, the PC is stored at `[esp]` in this example. Finally, backward traversal is performed starting at the seeding instruction. Hence instruction `fcmovb st(0), st(7)` is easily found, according to the method described above.

After constructing a chain of instructions through the two-way traversal, the execution of instructions in the chain is then emulated, as described below.

Detection of Self-modifying Decryption Routine. Self-modifying decryption routines are detected by performing emulated execution of the already found decryption instructions. The purpose of this execution is to determine the address to which the target instruction writes a value, or the address to which the target instruction branches, depending on the type of the target instruction. As far as the emulation is concerned, the decryption code of the input network traffic is mapped to a random virtual address space of the vulnerable program that the exploit code targets.

The emulation is limited in the following way. Instruction emulation proceeds until a decryption loop is detected, or an illegal instruction is encountered. If a memory location is modified that is within the emulated address space of the code, this fact is noted. It is evidence for the existence of a decryption routine. If the address of the target instruction branches points to the flow itself, the forward traversal is continued, otherwise it is stopped.

In a favorable situation, emulated instruction execution only occurs for a small number of instructions. This is because execution ends once a self-modifying decryption loop is revealed. For a decryption loop not using self-modifying techniques, only one traversal of the loop is needed to stop execution.

3.3 Decryption Routine Verification

The previous phase detects the presence of a possible decryption routine by finding the loops in its control flow structure. During the detection of the loop, a form of *GetPC* code should be available to find a pointer to the encrypted payload. The data flow of the detected loop is analyzed to improve the overall accuracy of the method. Two properties of of decryption routines are exploited for this purpose.

The first property is that in a detected loop, there must be a memory-write instruction that uses indirect addressing. That is, a register is used to contain an offset that partly identifies the location where data is to be read or written. In addition, the memory address pointers to the input network traffic. IA-32 [3] offers 24 memory addressing modes which can be classified into two categories - the direct and indirect addressing. For direct addressing, the memory operand's address is specified directly in the instruction. For indirect addressing, the memory operand's address is referenced through one or two registers. These registers offer a base address(stored in the base register) w/o an offset value(stored in the index register). A memory-write instruction using direct addressing is unlikely the instruction that directly modifies the encrypted payload. The hard-coded address easily results a fragile exploit code. (That is why the *GetPC* code is needed). For instance, the instruction at address `000D` in figure 2 (a) is such an example. IA-64 architecture also supports `RIP/EIP`-relative data addressing. The memory address can be referred through `RIP/EIP` registers.

The second property is that the register holding the address or offset must be updated within the loop. Otherwise the same memory location will be written over and over. In our current prototype, we only look for instructions that will update the register value in predictable and regular ways. For instance, `inc/dec/sub/add` instructions are most favorable for updating the registers. Other instructions, such as string instruction `lods` and loop instruction `loop` may also be used to update the register which holds the address or the offset. Future work will generalize this analysis. A possible way for the attackers to achieve the randomness is using a sequence of `push` instructions to specify the decryption order in the stack. The decryption loop then uses `pop` to get the order and then decode iteratively.

A few implementation details are as follows. Each instruction in a cycle is inspected to determine if it satisfies the first property, according to its opcode and addressing mode. If it is such an instruction, the cycle is cut to create an instruction sequence, with this instruction at the end. Then, other instructions in the sequence are sliced out by checking whether they have a data-flow dependency on the target instruction, using backward data flow analysis.

For example, suppose an unwrapped cycle contains the instruction sequence `inc eax, xchg eax,esi, xor [esi],ebx`. The first two instructions are sliced out because of their effect on register `esi`, used in the final instruction.

For checking the data flow dependency of two instructions, instructions are first converted, through into a semantics-preserving transformation, into an intermediate representation. This is helpful for overcoming code obfuscation techniques used in metamorphic exploits. For instance, a well crafted decryption routine may combine several processing steps into a single instruction. The `loop` and `lodsd` instructions shown in figure 2 are examples.

4. EVALUATION

A prototype of the proposed method has been implemented, and evaluated under realistic conditions. The results are described below.

4.1 Detection Rate

We tested the detection capability of the proposed approach on polymorphic exploits. These exploits were generated by two off-the-shelf polymorphic engines: ADMmutate [5], and Clet [6]. These engines have been used in other research papers for the same purpose [16, 14, 19, 18]. Also tested were polymorphic exploits generated by the Metasploit Framework [4]. This is a powerful open source framework for the construction and execution of exploits. This framework has also been used in other research [10, 19, 23].

The first experiment was as follows. 10 exploits were downloaded from Milw0rm (http://www.milw0rm.com). For each exploit, 10 polymorphic instances were generated, using the above tools (ADMmutate and Clet). ADMmutate may be the first well-known polymorphic engine. It can generate a simple metamorphic NO-OP sled with one-byte instructions, and a metamorphic decryption routine using several advanced obfuscation techniques. These include using multiple code paths for an operational instruction and inserting non-operational “junk” instructions. Clet can generate a metamorphic NO-OP sled using English words. It also uses “cramming” bytes to make the byte frequency of the resulting polymorphic exploit codes resemble that of normal network traffic.

Each of these exploit instances was then input to the proposed detection method. All 100 instances were successfully identified as exploit code. Both of these polymorphic engines generate encrypted exploit codes with an obvious NO-OP sled of sufficient length, as well as an obvious decryption loop. Previously-proposed detection methods [10, 23, 19, 22, 9] may also be able to detect such exploits. The existence of a sufficiently long NO-OP sled will help them cope with the non-obvious starting location of the decryption routine.

The second experiment simulated remote exploit attacks, using the Metasploit Framework. The target service was an unpacked Windows XP host running the Serv-U ftp server v4.0. Attacks were launched from a Windows host using polymorphic exploits for the following vulnerabilities:

- Serv-U FTPD MDTM Overflow [3]
- Microsoft RPC DCOM MS03-026 [4]
- Microsoft LSASS MSO4-011 Overflow[5]
- Microsoft ASN.1 Library Bitstring Heap Overflow [6]

For each vulnerability, we launched multiple attacks from the Metasploit console interface, using the following encoders (encryption methods):

1. `Pex`
2. `PexFnstenvSub`
3. `PexFnstenvMov`
4. `Countdown`

[3]http://www.osvdb.org/4073
[4]http://www.osvdb.org/2100
[5]http://www.osvdb.org/5248
[6]http://www.microsoft.com/technet/security/bulletin/MS04-007.mspx

5. `JmpCallAdditive`

6. `Alpha2`

7. `ShikataGaNai`

These were combined with two NO-OP sled generation methods: `Pex`, and `Opty`. `Pex` generates a NO-OP sled with one-byte instructions. `Opty` generates a NO-OP sled with multiple-byte instructions, as well as a "trampoline" sled, which transfers control using relative addressing directly to the exploit code. The traffic capture tool Ethereal was used to capture the network traffic generated by Metasploit. This traffic was then input to the prototype implementation of the proposed detection method.

The proposed approach successfully detected all of the polymorphic exploits generated using encoders `Pex`, `PexFnstenvSub`, `PexFnstenvMov`, `Countdown`, and `JmpCallAdditive`. These encoders generate static decryption code with the properties identified in section 3.3. They do not employ self-modification on decryption routines. Figure 2, for example, shows the disassembly of the decryption code produced by the `Countdown` and `JmpCallAdditive` encoders. The underlines mark the major functional decryption instructions: the seeding instruction of the *GetPC* code, the memory-write instruction for decrypting the encoded payload and the instruction for updating the address of encoded byte.

More impressively, the proposed method successfully detected 100% of the exploits generated by the `Alpha2` and `ShikataGaNai` encoders. These methods generate *self-modifying* decryption routines. The decryption loop is changed or patched "on the fly" (during execution) before it is used to decrypt the exploit. For illustration, figure 3 shows the disassembly of the self-modifying decryption code for `ShikataGaNai` encoder. Figure 3(a) shows the original decryption routine before execution. Figure 3(b) demonstrates the results after execution of the self-modifying decryption routine. The underlined instructions in (b) have the same effects as those shown in figure 2. In addition, the underlined bytes identify the modified instructions before and after execution. In the appendix, we also present the disassembly results of the self-modifying decryption routine for the `Alpha2` encoder.

The polymorphic exploit code for attacking Serv-U FTPD MDTM Overflow vulnerabilities do not use a NO-OP sled. This has been verified by inspection of the outputs generated under different configurations, and by inspection of the Metasploit source code. The absence of a NO-OP sled will likely defeat several proposed methods which specifically look for NO-OP sled [22, 9]. The network-level emulation method (e.g., [19]) is also likely to have problems identifying the start of the decryption routine. One of its heuristic for performance optimization is to skip several bytes (e.g. 50 bytes) after a zero byte is detected at a byte offset. Without the compensation effect of the NO-OP sled, instructions of the decryption routine code could be missed by the method. Sigfree [23] cannot detect polymorphic exploit code generated by small-sized decryption routines, such as `Countdown`, as mentioned in [23]. It also cannot detect polymorphic exploits that use self-modifying decryption routines, such as the exploit codes generated by encoder `ShikataGaNai`. The method proposed by Chinchani *et al.* [10] also cannot detect polymorphic exploits with self-modifying decryption routines.

In summary, the proposed method achieves a 100% detection rate on polymorphic exploit code, with or without NO-OP sleds, and with or without self-modifying decryption routines. No previous method of static analysis has been able to achieve this. The emulation method [19] can deal with the polymorphic exploit code with self-modifying decryption routines. However they are not robust against those without NO-OP sleds.

4.2 False Positives

We also tested the proposed method on normal (non-exploit) network traffic, and on Windows binary executables. A detection method should indicate in both cases that the traffic does not contain exploits. Indicating otherwise is regarded as a false positive.

We collected network traffic for five days from users in our lab, engaging in normal activities. Most of the traffic was UDP, FTP, HTTP, SSL, and other TCP data packets. Among these packets, the number of FTP and TCP packets containing downloaded executables, video files, and streaming video was significant (>90%). Over 4 million packets were captured, with a total payload size of more than 5 GB.

The data payloads from these packets were extracted and presented to the proposed method for testing, a packet at a time. Most exploits are small (a few tens of bytes) and easily fit within a single packet. (We discuss the limitations of this approach in section 5.) A packet incorrectly identified as containing an exploit was a false positive. The false positive rate was calculated as:

(# of falsely identified packets) / (Total # of packets)

Windows executables were also analyzed to determine the ability of the proposed method to distinguish exploit code from non-exploit code. Executables in the C: \ windows \system32 directory of a machine running Microsoft Windows XP, service pack 2, were used for this purpose. The total size of these files was around 1 GB. For analysis, we packetized each executable into a sequence of packets, and analyzed each packet separately. The false positive rate was calculated as above.

The results were as follows. The false positive rate was 0.0126% for the case of Windows executables, and 0.0002% for the case of captured network traffic. Only 8 out of more than 4 million packets resulted in false identifications, or alerts. The packet contents were manually inspected to verify that they did not contain exploits.

4.3 Processing Cost

We also measured the running time cost of the core detection algorithm of the proposed approach. The standard `C` function `clock()` was used for this purpose. Pairwise `clock()` functions were inserted appropriately to embrace the target detection procedures. The elapsed time between the pairwise `clock()` measurements was collected and accumulated. In these experiments, network packets and Windows executables of various sizes, ranging from several bytes to millions of bytes in length, were analyzed. The processing speed of our method is calculated as the sizes of processed packets or files against the corresponding processing time. The experiments were performed on a machine running Microsoft Windows XP, service pack2, with a Pentium(R)D 3.00GHz CPU, and 2GB of RAM. Figure 4 shows the normalized results.

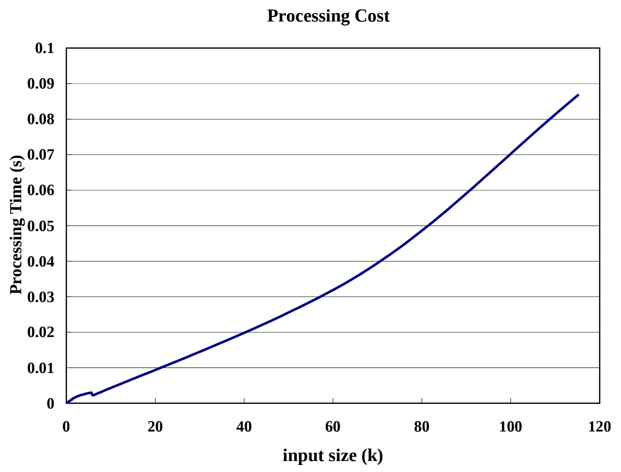


Figure 4: Running Time Overhead

Our non-optimized implementation demonstrates a modest processing speed. The results show almost a linear relationship between the packet size, and running time. The current implementation achieves a speed of roughly 1.5M/s. This method has not been optimized yet, and substantial speedups should be possible.

5. ATTACK ANALYSIS

We discuss now the possibility of defeating the the proposed detection method.

Fragmentation. Decryption routines are normally of small size. They can be contained within single packets. However, the attackers may deliberately split exploit traffic across multiple packets. It is trivial to reassemble packets before analysis, at the cost of modest additional processing overhead (i.e., the dependence on payload size is slightly greater than linear, as shown in Figure 4.)

No use of looping by the decryption routine. A loop is very likely to be needed for decryption purposes, since in-line coding of a decryption routine will otherwise be much longer (and therefore easier to identify). Interspersed in-line decryption code and the encrypted exploit payload should be highly carefully designed. This is because the decryption code after its working should be bypassed by the decrypted exploit code. Here we do not claim there are no such encryption or decryption methods. Instead, we speculate no use of looping for the decryption methods will raise the bar for the attackers.

Use of values not in the exploit code. Polychronakis *et al.* [19] have pointed out that attackers can use data from the environment in which the exploit executes. If self-modifying code relies on a key outside the address space of the exploit, this will not be detected by the proposed method at present. However, such exploits will be much more platform specific, and therefore much more sensitive to small system changes and randomization techniques [24].

Long or infinite loops. The analysis time of traversal and execution depends on the length of the derived chain of instructions. If the code contains a lengthy loop, or one which does not terminate, analysis may fail or may require an excessive amount of time. Nevertheless, our approach can still be useful as a first-stage detection method. Polychronakis *et al.* [19] demonstrated that long loops in normal network traffic are rare.

6. CONCLUSIONS

In this paper, we presented a new method for detection of self-decrypting polymorphic exploits. The proposed method scans network traffic for the presence of a decryption routine, which is characteristic of such exploits. The proposed method outperforms previous proposals [10, 23, 19, 22, 9] in its capability to identify more precisely the starting location of the decryption routine, with fewer assumptions. The method also can identify the decryption routine even if self-modifying code has been used to conceal its presence.

The evaluation results show that the proposed method has a 100% detection rate on realistic exploits of many types, including those that use self-modifying code, and/or that do not have a NO-OP sled. On a large collection of network traffic and Windows executables, a very low false positive rate was observed. The non-optimized implementation running time is roughly linear in the amount of data processed. These results indicate the proposed method is likely to be useful as part of an automated network defense again both targeted attacks, and large-scale zero-day worm outbreaks.

Future work will focus on generalizing the method for less obvious sequences of byte decoding. In addition, we will test the method on non-exploit code that uses code obfuscation, code encryption, and self-modification for legitimate purposes (e.g., to prevent reverse-engineering, and to protect license verification). We expect the way these techniques are used to be substantially different than exploit code.

7. REFERENCES

[1] Common vulnerabilities and exposures. http://cve.mitre.org/cve/downloads/full-cve.csv.

[2] Computer Economics. http://www.computereconomics.com.

[3] Intel Architecture Software Developers Manual. Volume 2: Instruction Set Reference.

[4] Metasploit project. http://www.metasploit.org.

[5] The ADMmutate polymorphic engine. http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz.

[6] The CLET polymorphism engine. http://www.phrack.org/show.php?p=61&a=9.

[7] Bro Intrusion Detection System, 2003. http://www.bro-ids.org.

[8] Snort: an open source network intrusion prevention and detection system, 2005. http://www.snort.org.

[9] P. Akritidis, E. Markatos, M. Polychronakis, and K. Anagnostakis. STRIDE: Polymorphic Sled Detection through Instruction Sequence Analysis. In *Proceedings of the 20th IFIP International Information Security Conference (SEC'05)*, pages 375–392, June 2005.

[10] R. Chinchani and E. Berg. A Fast Static Analysis Approach To Detect Exploit Code Inside Network Flows. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID'05)*, pages 284–308, September 2005.

[11] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-Aware Malware Detection. In *Proceedings of 2005 IEEE Symposium on Security and Privacy (S&P'05)*, pages 32–46, May 2005.

[12] J. C Foster and M. Price. *Sockets, Shellcode, Porting, & Coding: Reverse Engineering Exploits and Tool Coding for Security Professionals*. Syngress Publishing, USA, 2005.

[13] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static Disassembly of Obfuscated Binaries. In *Proceedings of the 13th USENIX Security Symposium*, pages 255–270, Auguest 2004.

[14] Z. Li, M. Sanghi, Y. Chen, M. Kao, and B. Chavez. Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. In *Proceedings of 2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 32–47, May 2006.

[15] S. S. Muchnick. *Advanced Comiler Design Implementation*. Morgan Kaufmann Publisher, CA, USA, 1997.

[16] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *Proceedings of 2005 IEEE Symposium on Security and Privacy (S&P'05)*, pages 226–241, May 2005.

[17] J. Newsome, B. Karp, and D. Song. Paragraph: Thwarting Signature Learning By Training Maliciously. In *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID'06)*, September 2006.

[18] U. Payer, M. Lamberger, and P. Teufl. Hybrid engine for polymorphic code detection. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment(DIMVA'05)*, pages 19–31, July 2005.

[19] M. Polychronakis, K. Anagnostakis, and E. Markatos. Network-Level Polymorphic Shellcode Detection Using Emulation. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment(DIMVA'06)*, July 2006.

[20] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *Proceedings of the 22th Annual Computer Security Applications Conference (ACSAC'06)*, December 2006.

[21] S. Sidiroglou and A. Keromytis. Countering Network Worms Through Automatic Patch Generation. In *Research Report*, 2003.

[22] T. Toth and C. Kruegel. Accurate Buffer Overflow Detection via Abstract Payload Execution. In *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID'02)*, pages 274–291, October 2002.

[23] X. Wang, C. Pan, P. Liu, and S. Zhu. SigFree: A Signature-free Buffer Overflow Attack Blocker. In *Proceedings of the 15th USENIX Security Symposium*, pages 225–240, July 2006.

[24] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Proceedings of the 22th International Symposium on Reliable Distributed Systems (SRDS'03)*, pages 260–269, October 2003.

[25] V. Yegneswaran, J. Giffin, P. Barford, and S. Jha. An architecture for generating semantic-aware signatures. In *Proceedings of the 14th USENIX Security Symposium*, pages 97–112, August 2005.

APPENDIX

Appendix

In this appendix, we provide the disassembly results of the self-modifying decryption routine for `Alpha2` encoder. The underlined instructions in figure 5 (b) are the seeding instruction of the *GetPC* code, the memory-write instruction for decrypting the encoded payload and the instruction for updating the address of encoded bytes. The underlined bytes in (a) and (b) highlight the contrast of modified instructions before and after execution.

```
0000  eb 03           jmp 0005                    0000  eb 03           jmp 0005
0002  59              pop ecx                     0002  59              pop ecx  (ecx=000A)
0003  eb 05           jmp 000A                    0003  eb 05           jmp 000A
0005  e8 f8 ff ff ff  call 0002                   0005  e8 f8 ff ff ff  call 0002
000A  49              dec ecx                     000A  49              dec ecx
000B  49              dec ecx                     000B  49              dec ecx
000C  49              dec ecx                     000C  49              dec ecx
000D  49              dec ecx                     000D  49              dec ecx
000E  49              dec ecx                     000E  49              dec ecx
000F  49              dec ecx                     000F  49              dec ecx
0010  49              dec ecx                     0010  49              dec ecx
0011  49              dec ecx                     0011  49              dec ecx
0012  49              dec ecx                     0012  49              dec ecx
0013  49              dec ecx                     0013  49              dec ecx
0014  48              dec eax                     0014  48              dec eax
0015  49              dec ecx                     0015  49              dec ecx
0016  49              dec ecx                     0016  49              dec ecx
0017  49              dec ecx                     0017  49              dec ecx
0018  49              dec ecx                     0018  49              dec ecx
0019  49              dec ecx                     0019  49              dec ecx
001A  49              dec ecx                     001A  49              dec ecx
001B  49              dec ecx                     001B  49              dec ecx
001C  51              push ecx                    001C  51              push ecx
001D  5a              pop edx                     001D  5a              pop edx
001E  6a 46           push 46                     001E  6a 46           push 46
0020  58              pop eax                     0020  58              pop eax
0021  30 42 31        xor [edx+31], al            0021  30 42 31        xor [edx+31], al
0024  50              push eax                    0024  50              push eax
0025  41              inc ecx                     0025  41              inc ecx   <--+
0026  42              inc edx                     0026  42              inc edx      |
0027  6b 42 41 56     imul eax, [edx+41],56       0027  6b 42 41 10     imul eax, [edx+41],10
002B  42              inc edx                     002B  42              inc edx      |
002C  32 42 41        xor al, [edx+41]            002C  32 42 41        xor al, [edx+41]
002F  32 41 41        xor al, [ecx+41]            002F  32 41 41        xor al, [ecx+41]
0032  30 41 41        xor [ecx+41], al            0032  30 41 41        xor [ecx+41], al
0035  58              pop eax                     0035  58              pop eax      |
0036  50              push eax                    0036  50              push eax     |
0037  38 42 42        cmp [edx+42],al             0037  38 42 42        cmp [edx+42],al
003A  75 5a           jne 0096                    003A  75 e9           jne 0025  ---+
003C  49              dec ecx                     003C  49              dec ecx
....  <encrypted payload>                         ....  <encrypted payload>

                 (a)                                               (b)
```

Figure 5: Disassembly of Self-Modifying decryption routine for Alpha2 encoder. a) Before Execution b) After Execution. In each figure, the leftmost column shows instructions' addresses represented in `hex` format; the middle column shows the actual instruction bytes; the rightmost column shows the decoded instructions. Instruction `call 0002` at address `0005` and `pop ecx` at address `0002` are the *GetPC* code of this example.