On Emulation-Based Network Intrusion Detection Systems

Ali Abbasi[1], Jos Wetzels[1,2], Wouter Bokslag[2],
Emmanuele Zambon[1,3], and Sandro Etalle[1,2]

[1] Services, Cyber security and Safety Group, University of Twente, The Netherlands,
{a.abbasi, emmanuele.zambon, sandro.etalle}@utwente.nl,
a.l.g.m.wetzels@student.utwente.nl
[2] Eindhoven University of Technology, The Netherlands,
s.etalle@tue.nl, w.bokslag@student.tue.nl
[3] SecurityMatters BV, The Netherlands
emmanuele.zambon@secmatters.com

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 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. We draw the conclusion that current emulation-based systems have limitations that allow attackers to craft generic shellcode encoders able to circumvent their detection mechanisms.

Keywords: Emulation, IDS, Shellcode, Evasion, Polymorphism

1 Introduction

Emulation-based Network Intrusion Detection Systems (EBNIDS) where introduced by Polychronakis et al.[1] to identify the presence of (possibly polymorphic) shellcode in network communication. The original motivation for introducing a new kind of NIDS was to overcome the limits of signature-based NIDS, which by definition can only identify known shellcodes, and are easily circumventable, e.g., by using polymorphism.

The main idea behind EBNIDSes 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 signs of being malicious. EBNIDSes work by turning the payload of a suspected network flow into a sequence of instructions and by simulating these instructions to determine what

they actually do. The resulting behavior is then analyzed with the help of specific heuristics.

After their introduction, we have seen a growing interest in this field, with a number of new proposals being introduced in a relatively short time-span [2–7].

The goal of this paper is to investigate the actual practical effectiveness of EBNIDSes. In particular, in this paper

- we illustrate how EBNIDSes work by introducing three abstraction layers that allow us to describe all the approaches proposed so far,
- we investigate and question the actual effectiveness of EBNIDS, by providing evidence that present EBNIDSes have intrinsic limitations that make them evadable using standard coding techniques.

To substantiate the second point, we introduce simple coding techniques exploiting the implementation and/or design limitations of EBNIDSes, and show that they allow attackers to completely evade state-of-the-art EBNIDSes. Finally, we prove that it is possible to write a shellcode that evades EBNIDSes even in presence of a (theoretical) more complete implementation of the pre-processor and the emulator. In particular, we show it is still possible to evade both the emulation phase and the heuristics engine of EBNIDSes. These evasion techniques do not leverage implementation bugs of EBNIDSes (e.g., instruction set support) but exploit limitations in the concept of emulation and in the design of heuristics detection patterns.

Here we want to stress that we do not include in the research those intrusion detection systems relying on a precise memory image of the target, like Argos [7], because they are intrinsically different from EBNIDSes; indeed they are considered *host-based* (rather than *network-based*) NIDSes.

2 Detecting shellcode on Emulation based NIDS

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.

2.1 Pre-processing

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 which 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

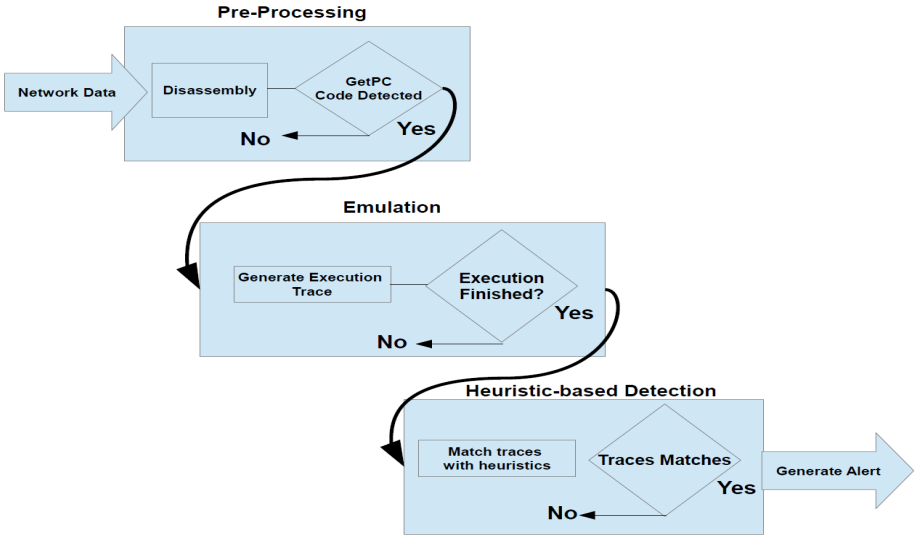

Fig. 1. Overview of Emulation Based Intrusion Detection System functionalities

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.

2.2 Emulation

The emulation step 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. In order to allow inspecting traffic in real-time, emulation is constrained by execution time, which imposes compromises on the implementation of emulators. Software-based emulators generally only support a subset of all hardware supported instructions for a restricted amount of hardware architectures. As an example, the approaches proposed by Polychronakis et. al. in [1, 9] support a subset of x86 instructions, which do not include floating point (FPU), MMX, and SSE instructions which are commonly available in modern CPUs or GPUs. In addition, the emulator does not know about the execution environment of the potential target of the shellcode (i.e., the machine on which the shellcode could run). For these reasons, it is not always possible to reliably emulate all shellcodes. To overcome

this problem Polychronakis et. al. propose to employ a generic memory image [3]. By means of the generic memory image the emulator can read and jump to generic data structures and system calls, but still without guarantee that the values present at certain locations in memory will correspond to the values in the target memory.

2.3 Heuristic-based detection

The heuristic-based detection step consists of examining the execution trace produced by the emulator searching for known patterns of shellcode execution. If such patterns are found, the suspected network data is flagged as a shellcode and an alert can be raised by the EBNIDS. Three basic heuristics have been proposed over time to identify patterns of polymorphic shellcode in execution traces (see[1, 9]):

1. **GetPC code**: any shellcode must at some point obtain its own address in memory to read its own body and get environmental information, since such information can not be known prior to execution. This procedure is known as GetPC code. In its simplest form, the GetPC code consists of invoking CALL or FSTENV instructions. A heuristic to detect shellcode using GetPC code is built by searching for the GetPC seeding instructions and then ensuring that the execution trace of the code emulated starting from the GetPC instructions terminates.
2. **Payload read**: the decryption routine of polymorphic shellcode needs a large amount of memory accesses to read the encrypted payload. On the other hand, non-malicious code shows a limited frequency of unique memory reads. A heuristic to detect polymorphic shellcode is built by observing in an execution trace some form of GetPC code followed by a number of unique memory reads exceeding a so-called Payload Reads Threshold (PRT).
3. **WX instructions**: the decryption routine of polymorphic shellcode needs to write the decrypted instructions to memory. Executed instructions residing at memory addresses that were previously written are called WX instructions (write-execute instructions). A decrypted shellcode consists of such WX instructions, which may be allocated in a memory area different from the encrypted shellcode. A heuristic to detect polymorphic shellcode based on these observations consists of checking if, at the end of an execution trace, the emulator has performed W unique writes and has executed X WX instructions. In which case the payload is flagged as a non-self-contained polymorphic shellcode.

An extended set of heuristics is proposed in [3] to identify the presence of shellcode in arbitrary data streams. These runtime heuristics (which only cover Windows shellcodes) 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.

1. **Kernel32.dll base address resolution**: most shellcodes require interacting with the OS through the system call interface or user-level API. In order to call an API function, the shellcode must first find its absolute address in the address space of the process. Kernel32.dll provides two 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 recognizing two methods (using the Process Environment Block or Backwards Searching) of obtaining the Kernel32.dll base address. This particular heuristics focus on behavior specific to Windows shellcode.
2. **SEH-based GetPC code**: when an exception occurs, the system generates an exception record that contains the necessary information for handling it. In particular, the exception record contains the Program Counter (PC) value at the time the exception was triggered. This information is stored on the stack. A shellcode can register a custom exception handler, trigger an exception, and then extract the absolute memory address of the faulting instruction. Gene has a heuristic that detects any shellcode installing a custom exception handler, including polymorphic shellcode that uses SEH-based GetPC code.
3. **Process memory scanning**: some software vulnerabilities allow injecting only a limited amount of code, usually not enough for a fully functional shellcode. In most cases though, the attacker has the ability to deploy a second, much larger payload which will be stored at a random memory location (e.g., in a buffer allocated in the heap). The (first-stage) shellcode then needs to scan 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 address is valid and readable. Gene has a heuristic that recognizes shellcode attempting at retrieving information about paged memory through Structured Exception Handler (SEH) and syscall-based scanning methods.

3 Evading EBNIDSes

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[4] 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. Instruction decoding is handled by the *libdasm* disassembly library, while the emulation and heuristic-based detection steps are 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 described in Section 2. 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, instruction decoding is handled by the *libdasm* disassembly library, 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 Section 2, 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

[4] The authors plan to release all the implemented techniques as Metasploit plugins.

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 by the author 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 which 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.

3.1 Evasions exploiting implementation limitations

Limitations of the pre-processor implementation

In most EBNIDSes, static analysis is applied in the pre-processing step to determine which sequences of bytes should be emulated [2, 4, 8]. 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.

Although the authors state that self-modifying code or indirect addressing is unlikely to appear before the GetPC code (since this would require a base-address for referencing) we argue that this is not the case. First, it is possible for an attacker to construct the shellcode on the stack in a dynamic fashion, including the GetPC code. Secondly, the attacker can avoid GetPC seeding instructions altogether and construct the entire shellcode on the stack. This would require a full emulation for detection, since it would be unfeasible to detect GetPC seeding instructions contained in a self-modifying code statically, especially if instructions are encoded using a randomized key. In the absence of the capacity to detect seeding instructions, subsequent analysis will fail as well.

Based on these observations, we create a shellcode encoder which consists of XORing the shellcode with a random key and prepending a decoder armored with anti-disassembly GetPC code. To build the anti-disassembly GetPC code

we adapt four existing techniques proposed by Branco et. al. [13] and Sikorski et. al. [14] for preventing malware analysis:

1. *Use of garbage bytes and opaque predicates*: the insertion of garbage bytes after so-called opaque predicate instructions confuses some disassemblers into taking the bytes immediately after such an instruction as the starting point of a next instruction. Opaque predicates are logical tautologies or contradictions which are constructed in such a way that this can not be easily determined without evaluating them. For example, (GetUserID() xor 0x0A0A) is opaque for any instance evaluating it that does not know beforehand the result of GetUserID(), while an attacker can construct this when targeting user with id 0x0A0A specifically.
2. *Flow redirection to the middle of an instruction*: certain instructions are crafted to contain other instructions in the middle of their opcodes. During execution, the code flow is redirected to the middle of instructions to execute those "hidden" inside. This requires full emulation for proper disassembly.
3. *Push/pop-math stack-constructed shellcode*: instead of executing instructions directly, the 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.
4. *Code transposition*: a piece of code is split into separate parts and re-arranged 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.

We evaluate these anti-disassembly techniques against Nemu and Libemu by encoding our test shellcodes with the anti-disassembly encoder described above. If the anti-disassembly encoder works, the pre-processor cannot correctly identify the GetPC code and the shellcode analysis will stop without raising alerts.

	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. Anti-disassembly techniques detection rate

Table 1 shows the results of the tests. While we could bypass Libemu using Garbage bytes and Push/Pop math techniques, Nemu has better detection in most cases with the only successful evasion technique being Code Transposition and Push/pop math in one case. We believe this is due to the fact that Nemu did not properly disassemble all the instructions of our armored decoder. This impacts the emulation of such instructions, eventually preventing the correct execution of the decoding routine. As a result, the decoded shellcode cannot

be completely emulated and this causes the failure of heuristics requiring the observation of a large number of instructions to trigger. However, we consider this is an exceptional case, while in general we conclude that the evasion techniques were ineffective against Nemu.

Limitations of the emulator implementation

Unsupported instructions Most EBNIDSes do not provide full emulation capabilities and only emulate a subset of the full instruction set. For example, the approaches presented by Polychronakis et. al. in [1, 9] use *libdasm* as disassembler and implement a subset of the IA-32 instruction set, including most general purpose instructions but no FPU, MMX or SSE instructions.

It is possible for an attacker to construct a shellcode which incorporates instructions not covered by the limited emulators, therefore causing emulation to stop when such instructions are encountered, and therefore preventing the heuristic-based detection. Additionally, it is possible to use the results of non-emulated instructions as an integral part of a self-modifying routine.

In addition to emulating only a subset of the IA-32 instruction set, all emulators provide only a subset of the complete system functionality, including syscall emulation, virtual memory and the presence of process images. These limitations in the implementation of system functionality emulation can be abused by an attacker in order to thwart successful emulation and thus detection.

Based on these observations, we create a shellcode encoder which consists of XORing the shellcode with a random key and prepending a decoder made with instructions which are not supported by some types of emulators. In more detail, we create five versions of the decoder, each using different types of instructions:

1. FPU instructions (using FNSTENV).
2. FPU instructions (using FNSAVE).
3. MMX instructions.
4. SSE instructions.
5. Instructions considered obsolete or undocumented by some disassemblers and emulators such as salc or xlatb instructions.

We evaluate these anti-disassembly techniques against Nemu and Libemu by encoding our test shellcodes with the anti-disassembly encoder described above. If the encoder works, the emulator cannot correctly execute the GetPC code and the shellcode analysis will stop without raising alerts.

Table 2 summarizes the test results. With the only exception of the FNSTENV FPU instruction, all other instruction sets prevented the emulator to successfully emulate the decoder and detect the shellcodes.

Emulator detection Emulator detection refers to a class of techniques which 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

	FPU FN-STENV	FPU FN-SAVE	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. Unsupported instructions evasion detection rate

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 which 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 which 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 an 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. Table 3 shows an example of the running time difference between Nemu, Libemu and a native CPU when executing four different types of operations (see Section 3.2 for a description of the operations).

	Opaque instructions	Intensive loop	Integrated loop	RDA
Nemu	6.80	9.08	37.81	52.90
Libemu	44.07	75.20	173.49	177.56
native	0.148	2.10	0.30	0.68

Table 3. Difference in actual running time (in milliseconds) between Nemu, Libemu and native CPU.

We create a shellcode encoder which 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.

Limitations of existing heuristics

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

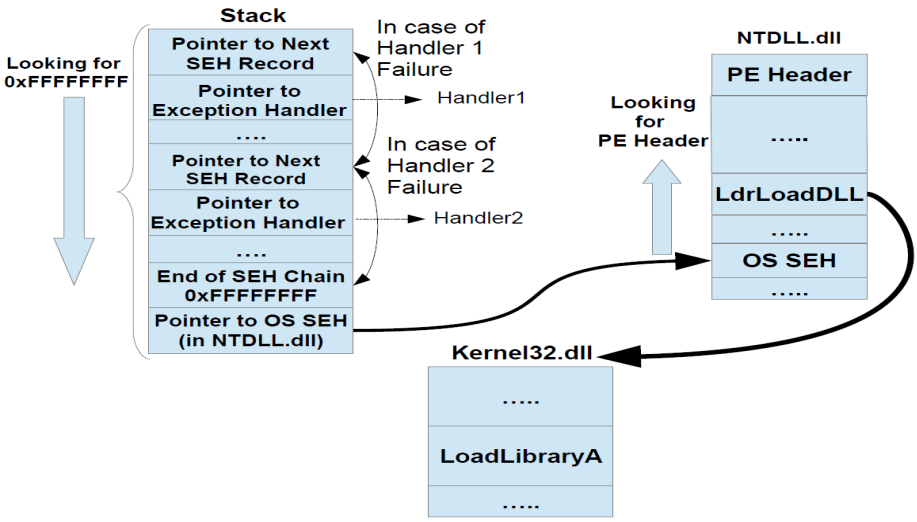

Fig. 2. SEH scanning technique for resolving the base address of Kernel32.dll

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).

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.

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.

Evasion of Payload read heuristics Nemu includes a heuristic for detecting self-modifying code called Payload Read Threshold (PRT). The heuristic consist of imposing a threshold on the number of unique read operations executed by the payload, combined with the presence of GetPC code. To circumvent this heuristic [17] proposes to use syscalls to execute read operations instead of reading directly in the payload shellcode. We implement a shellcode using this approach and notice that despite the fact that the technique has been public since 2009, the Nemu heuristic has not been updated to detect this technique. Note that Nemu has another heuristic, which imposes a threshold on the number of syscalls and could in principle trigger when this kind of evasion is used. However, the heuristic was designed to detect egg hunting, and as such the threshold imposed on the number of syscalls is way higher than the number of syscalls needed in the evasion payload. For this reason, even this second heuristic is ineffective against our implementation.

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.

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 cannot 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.

3.2 Evasions exploiting on intrinsic limitations

Limitations of the emulator

Fragmentation So-called Swarm or fragmentation attacks [20] are a class of attacks in which an attacker can 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 a designated location. After building the decoder in this fashion, the last attack instance hijacks the control of the attacked process to start the execution of the decoder while simultaneously including the shellcode cipher text. As such, swarm attacks could be considered a form of fragmented egg-hunting attacks. Swarm attacks can defeat all three components of EBNIDSes. It will be a severely complicated task to do static analysis for part of the decoder, in the 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 EBNIDSes but have the downside of being applicable only in specific exploitation scenarios (e.g., the application must keep all the different pieces of the shellcode in memory until the last piece of the shellcode is sent). Because of this we could not easily build a test to evaluate this evasion technique.

Limitations of faithful emulation

Non-self-contained shellcode 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 which 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 IDSes, for EBNIDSes it is unscalable to keep up-to-date information about all possible target hosts in a network. The approaches in [1, 9, 8, 2–4] 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, or by resolving the library base address by using the SEH walk technique described in Section 3.1 . 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.

Execution threshold Real-time intrusion detection imposes the need to evaluate whether input is malicious or not within a reasonable amount of time. Shellcodes which 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 which 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:

1. **Opaque loops**: 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 used 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.
2. **Intensive loops**: 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.
3. **Integrated loops**: 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 which 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.
4. **Random Decryption Algorithm**: in this technique the payload is encrypted with a random key. The shellcode attempts to bruteforce the key and, after each attempt, checks the decrypted body against a hash value. The original RDA implementation [24] still needed plaintext GetPC code to know the address of the encrypted payload body. In our implementation we generate a second RDA key, XOR the GetPC instructions with the key and modify the decoder to first decrypt the GetPC as well.

Table 4 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 be 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.

	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 4. Detection of execution threshold evasion techniques against Libemu and Nemu

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 [27]. EBNIDS approaches [1, 9, 3, 8, 2, 4] 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, the Metasploit CKPE encoder can be detected by EBNIDSes 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 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.

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 an (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 unarmoring 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 unarmoring 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.

4 Conclusions and Future Works

In this paper, 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 heuristic-based detection can be easily bypassed by a skilled attacker. We have shown that it is possible to write generic shellcode encoders which 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.

It is not in the scope of this paper to investigate how to mitigate these problems. We believe that promising avenues of research are those dealing with algebraic specification, hidden Markov-Models and neural networks. Regarding the limitations related to the incompleteness of the emulation environment but there has been research into the detection of ROP attacks such as [29–31] which would be crucial for any network intrusion detection system to implement.

Acknowledgement The work of the second author has been partially supported by the dutch government national AVATAR project. The work of the fourth author has been partially supported by the European Commission through project FP7-SEC-285477-CRISALIS funded by the 7th Framework Program. The work of the fifth author has been partially supported by the European Commission through project FP7-SEC-607093-PREEMPTIVE funded by the 7th Framework Program.

References

1. M. Polychronakis, K. Anagnostakis, and E. Markatos, "Network–Level polymorphic shellcode detection using emulation," in *Detection of Intrusions and Malware & Vulnerability Assessment.* Springer, 2006, pp. 54–73.
2. M. Shimamura and K. Kono, "Yataglass: Network-level code emulation for analyzing memory-scanning attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment.* Springer, 2009, pp. 68–87.
3. M. Polychronakis, K. Anagnostakis, and E. Markatos, "Comprehensive shellcode detection using runtime heuristics," in *Proc. of the 26th Annual Computer Security Applications Conference (ACSAC10).* ACM, 2010, pp. 287–296.
4. K. Snow, S. Krishnan, F. Monrose, and N. Provos, "SHELLOS: Enabling Fast Detection and Forensic Analysis of Code Injection Attacks," in *USENIX Security Symposium*, 2011.
5. B. Gu, X. Bai, Z. Yang, A. Champion, and D. Xuan, "Malicious shellcode detection with virtual memory snapshots," in *Proc. of IEEE INFOCOM 2010.* IEEE, 2010, pp. 1–9.
6. M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda, "Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment.* Springer, 2009, pp. 88–106.
7. G. Portokalidis, A. Slowinska, and H. Bos, "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, no. 4. ACM, 2006, pp. 15–27.
8. Q. Zhang, D. Reeves, P. Ning, and S. Iyer, "Analyzing network traffic to detect self-decrypting exploit code," in *Proc. of the 2nd ACM symposium on Information, Computer and Communications Security (CCS07).* ACM, 2007, pp. 4–12.
9. M. Polychronakis, K. Anagnostakis, and E. Markatos, "Emulation-based detection of non-self-contained polymorphic shellcode," in *Proc. of Recent Advances in Intrusion Detection (RAID07).* Springer, 2007, pp. 87–106.
10. Honeynet Project, "Dionaea, a low-interaction honeypot," http://www.honeynet.org/project/Dionaea, 2008.

11. E. Markatos and K. Anagnostakis, "Noah: A european network of affined honeypots for cyber-attack tracking and alerting," *The Parliament Magazine*, no. 262, 2008.
12. P. Baecher and m. Koetter, "libemu," http://libemu.carnivore.it/, 2009.
13. R. Branco, G. Barbosa, and P. Neto, "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. M. Sikorski and A. Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, 2012.
15. P. Ferrie, "Attacks on more virtual machine emulators," *Symantec Technology Exchange*, 2007.
16. T. Raffetseder, C. Kruegel, and E. Kirda, "Detecting system emulators," in *Information Security*. Springer, 2007, pp. 1–18.
17. P. Bania, "Evading network-level emulation," *arXiv preprint arXiv:0906.1963*, 2009.
18. Skape, "Using dual-mappings to evade automated unpackers," http://www.uninformed.org/?v=10&a=1&t=sumry, October 2008.
19. C. Linn, M. Rajagopalan, S. Baker, C. Collberg, S. Debray, and J. Hartman, "Protecting against unexpected system calls," in *Proc. of the 14th USENIX Security Symposium*, 2005, pp. 239–254.
20. S. Chung and A. Mok, "Swarm attacks against network-level emulation/analysis," in *Proc. of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID08)*. Springer, 2008, pp. 175–190.
21. 0vercl0k, "RP++ ROP Sequences Finder," https://github.com/0vercl0k/rp, 2013.
22. kingcopes, "Attacking the Windows 7/8 Address Space Randomization," http://kingcope.wordpress.com/2013/01/24/attacking-the-windows-78-address-space-randomization/, 2013.
23. M. Polychronakis and A. D. Keromytis, "Rop payload detection using speculative code execution," in *Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on*. IEEE, 2011, pp. 58–65.
24. Kharn, "Exploring RDA," http://www.awarenetwork.org/etc/alpha/?x=3, 2006.
25. R. Rivest, A. Shamir, and D. Wagner, "Time-lock puzzles and timed-release crypto," Massachusetts Institute of Technology, Tech. Rep., 1996.
26. Nomenumbra, "Countering behavior based malware analysis," https://har2009.org/program/track/Other/57.en.html, 2009.
27. D. Glynos, "Context-keyed Payload Encoding: Fighting the Next Generation of IDS," in *Proc. of Athens IT Security Conference (ATH.C0N 10)*, 2010.
28. J. Aycock, R. de Graaf, and M. J. Jr., "Anti-disassembly using cryptographic hash functions," *Journal in Computer Virology*, vol. 2, no. 1, pp. 79–85, 2006.
29. L. Davi, A. Sadeghi, and M. Winandy, "ROPdefender: A detection tool to defend against return-oriented programming attacks," in *Proc. of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS11)*. ACM, 2011, pp. 40–51.
30. P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie, "DROP: Detecting return-oriented programming malicious code," in *Information Systems Security*. Springer, 2009, pp. 163–177.
31. K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-Free: defeating return-oriented programming through gadget-less binaries," in *Proc. of the 26th Annual Computer Security Applications Conference (ACSAC10)*. ACM, 2010, pp. 49–58.