Emulation-based Software Protection

William Kimball (wkimball@afit.edu)

Black Hat DC 2009

This document provides an overview of two emulation-based software protection schemes which provide protection from reverse code engineering (RCE) and software exploitation using encrypted code execution and page-granularity code signing, respectively. The two protection mechanisms execute within trusted emulators while remaining out-of-band of untrusted systems being emulated. The integrity and reliability of the protection mechanisms depend upon attackers remaining sandboxed within the emulated environments. The three sections below provide an overview of emulation sandboxing, emulation-based encrypted code execution and emulation-based page granularity code signing.

1 Emulation Sandboxing

1.1 Problem Statement

Kernel malware is able to modify (attack) kernel protection mechanisms.

1.2 Approach

This protection scheme assumes it is more difficult to *break out*[1] of an emulation-based sandbox than break out of user-mode (Ring 3) to kernel-mode (Ring 0). Given the case, protection mechanisms implemented out-of-band of an emulated environment are less frequent to attack than kernel protection mechanisms. The following describes the emulation sandbox model.

1. Host OS copies Guest OS instructions from Guest OS memory into Host OS memory. (This step is needed for the protection mechanisms in the following sections.)
2. Guest OS instructions are translated (or interpreted) to a set of Host OS instructions. When this set of translated Host OS instructions execute the state of Guest OS memory and registers is modified such that it appears as if the original Guest OS instructions had been executed.
3. The translation process ensures Guest OS instructions read and write Guest OS memory exculsively. Host OS memory is inaccessible by Guest OS instructions and the translated

[1] Breaking out of an emulation-based sandbox refers to reading, writing or executing memory not allocated for the emulated environment.

set of Host OS instructions can not read or write Host OS memory.[2] As a result, the set of translated instructions will never be self-reading. The Guest OS instructions sandbox is the restricted memory space.

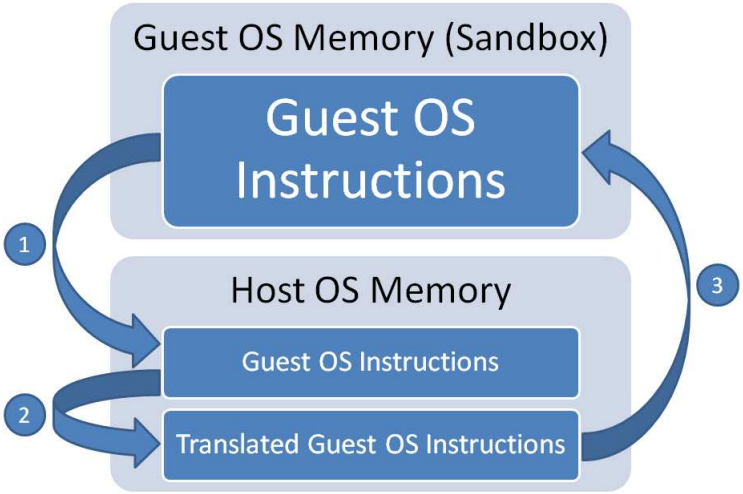

Figure 1: Basic Emulator Sandbox

2 Emulation-based Encrypted Code Execution

2.1 Problem Statement

Reverse Code Engineering (RCE) uncovers the internal workings of a program. It is used during vulnerability and intellectual property (IP) discovery. To protect from RCE program code may have anti-disassembly, anti-debugging, and obfuscation techniques incorporated. These techniques slow the process of RCE, however, once defeated protected code is still comprehensible. As an alternative, code may be encrypted. While encryption provides static code protection, encrypted code must be decrypted before execution.

2.2 Approach

This protection scheme keeps code encrypted within the emulated environment *during execution*. Decryption routines and secret keys remain out-of-band of the emulated environment.

1. Host OS copies encrypted Guest OS instructions from Guest OS memory into Host OS memory. The encrypted Guest OS instructions are decrypted in Host OS memory. The decrypted instructions always remain out-of-band of the Guest OS and are not accessible by Guest OS instructions.

[2]Given a properly implemented emulator with no design or implementation errors.

2. Decrypted Guest OS instructions are translated (or interpreted) to a set of Host OS instructions. When this set of translated Host OS instructions execute the state of Guest OS memory and registers is modified such that it appears as if the original Guest OS instructions had been executed.

3. The translation process ensures Guest OS instructions never read decrypted Guest OS instructions. The encrypted instructions are decrypted in Host OS memory using Host OS routines. Since the emulation sandbox ensures Host OS memory is inaccessible by Guest OS instructions, through emulation sandboxing, the decrypted instructions and decryption routines remain out-of-band and thus less frequent to attack.

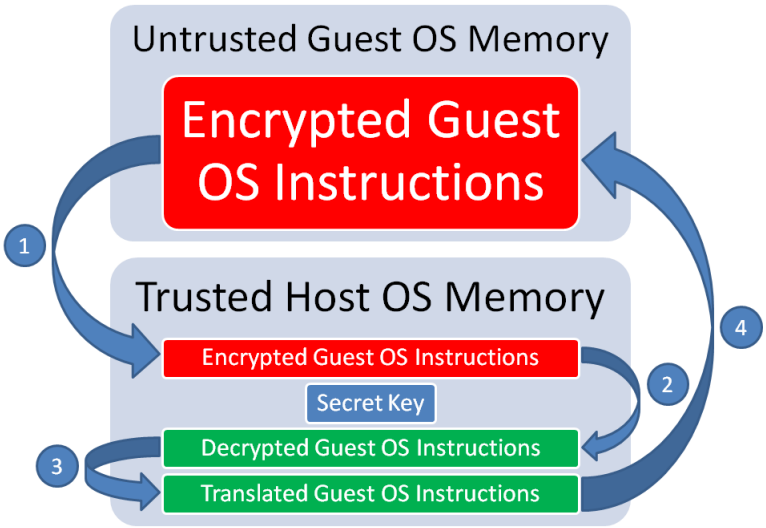

Figure 2: Encrypted Code Execution

2.3 Encrypted Code Execution using SecureQEMU

Our implementation of this protection scheme is called SecureQEMU and is based on a modified Quick Emulator [2]. Figure 3 shows Windows Notepad after encrypted using SecureQEMU. Notice how Notepad's encrypted instructions, shown in IDA, is exactly the same as the encrypted instructions shown in OllyDbg and WinDbg (arbitrary of privilege level). These instructions will never show decrypted in the debugger (or anywhere in the Guest OS's memory), even when the instructions are executing. Furthermore, the encrypted executable does not contain decryption routines and does not contain the key needed to decrypt the instructions.[3]

[3]Unlike other encryption-based software protections, SecureQEMU does not obfuscate the decryption process or use contextual keying. Decryption routines and keys remain out-of-band.

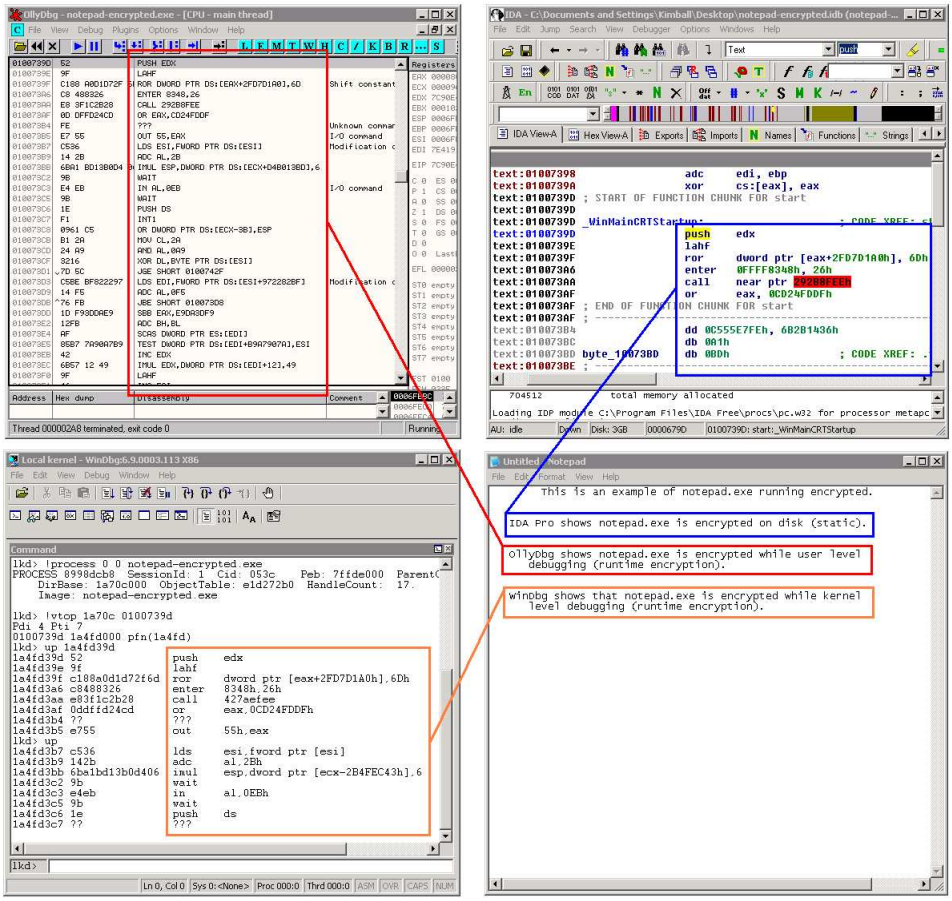

Figure 3: Notepad Remaining Encrypted During Execution

3 Emulation-based Page Granularity Code Signing

3.1 Problem Statement

Software exploitation is a process that leverages design and implementation errors (i.e. buffer overlows, input-driven format strings, integer overflows, race conditions, etc.) to cause unintended behavior which may result in security policy violations. Traditional exploitation protection mechanisms (i.e. stack canaries, variable reordering, shadow arguments, SafeSEH, NX pages, link-pointer sanity checking, pointer encoding, heap cookies, ASLR, etc.) provide a blacklist approach to software protection. Specially crafted exploit payloads bypass these protection mechanisms.

3.2 Approach

This protection scheme provides a *whitelist* approach to software protection by executing *signed code* exclusively. Unsigned malicious code (exploits, backdoors, rootkits, etc.) remain unexecuted,

therefore, protecting the system.

1. Host OS copies Guest OS instructions and Hash Message Authentication Code's (HMAC) (or digital signatures) of Guest OS instructions from Guest OS memory into Host OS memory.
2. HMACs of Guest OS instructions are recomputed using a secret key in Host OS memory. The secret key remains in Host OS memory and is never accessible by Guest OS instructions. Guest OS instructions with valid HMACs are translated (or interpreted) to a set of Host OS instructions. This set of Host OS instructions is executed as before.
3. Guest OS instructions with invalid HMACs remain untranslated and therefore unexecuted. Malicious code (unless signed using the secret key) will remain unexecuted, thus protecting the system.

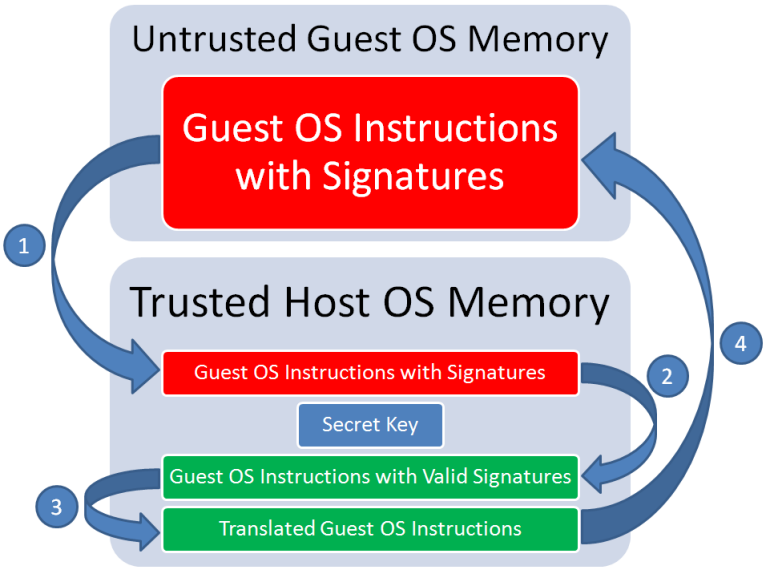

Figure 4: Page Granularity Code Signing

References

[1] S. Burnett and S. Paine, *The RSA Security's Official Guide to Cryptography.* Berkeley, CA, USA: Osborne/McGraw-Hill, 2001.

[2] F. Bellard, "Qemu, a fast and portable dynamic translator," in *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference.* Berkeley, CA, USA: USENIX Association, 2005, pp. 41–41.