Digital Protection
Editors: Michael Lesk, lesk@acm.org
Martin R. Stytz, mstytz@att.net
Roland L. Trope, roland.trope@verizon.net

Software Protection through Anti-Debugging

Michael N. Gagnon, Stephen Taylor, and Anup K. Ghosh
George Mason University

You're a software engineer, and you've developed a great copy-protection scheme that will save your company millions in lost revenue due to piracy. You know your protection scheme will work fantastically—as long as hackers don't circumvent it. Or perhaps you're a malware analyst, and you've discovered a worm that propagates slowly across networks, but you're not sure what else it might be doing. Your company's antivirus software product depends on you to reverse engineer the worm code to stop it. If you break the code first, you will stake your claim to fame.

These two scenarios illustrate the yin and yang of reverse engineering and anti-reverse engineering. In the first, hackers want to reverse engineer your software as the first step to cracking your copy-protection scheme, which you naturally want to prevent. In the second scenario, if you can't reverse engineer the worm quickly, it might spread widely, causing unknown damages. In one case, it's imperative to prevent reverse engineering to stop software piracy and the loss of revenue. On the flip side, reverse engineering software is critical for defending computer networks. Reverse engineering techniques, as well as those used to defeat reverse engineering, give scientists, engineers, and hackers alike hope that they can protect or copy software, and defend against or proliferate malicious attacks.

This column presents the reverse-engineering battle from an anti-debugging perspective. Anti-debugging encompasses the strategies, techniques, and tricks that protected software uses to attack debuggers and thwart reverse engineering. Current research on anti-reverse engineering typically focuses on obfuscation, virtual machine detection, antidisassembly, and tamper proofing,[1] but little attention has been paid to anti-debugging—an important tool in the software protection arsenal. This lack of attention is surprising because anti-debugging is frequently employed in practice by malicious code writers as well as in copy-protection techniques.

We focus here on describing state-of-the-art attacks on debuggers to prevent reverse engineering. You can use the information we present as part of your strategy to protect your software or to assist you in overcoming the anti-debugging tricks present in malicious software.

Detection

To understand anti-debugging, it's important to first understand that a debugger isn't a solitary and independent system. Rather, debugging systems are composed of multiple, layered debugging subsystems that collectively facilitate debugging. Each debugging subsystem provides a different level of support; the canonical debugging system is composed of a hardware layer beneath a software layer, and a not-to-be-underestimated human layer.

The CPU provides the hardware debugging subsystem—for example, the Intel x86 architecture supports debugging through tracing support, debug registers, and special interrupt instructions. Software debuggers such as GDB (GNU Debugger), OllyDbg, and SoftICE provide an interface that links the hardware subsystem and the human analyst.

Anti-debugging tricks work by detecting or exploiting specific debugging subsystems. Software that employs anti-debugging techniques can determine if it's being debugged by identifying artifacts—side effects of the debugging process—whether from the hardware, software, or human layers. Software can passively observe or actively invoke artifacts. For example, software processes can detect hardware debugging artifacts when a hardware debugging subsystem uses debug registers to place breakpoints on processes. The software process needs only to check debug registers for specific values.[2] Detecting debugging through a debug register (shown in Figure 1) is an example of passive observation of a hardware debugging artifact. Processes can also actively produce unintended side effects from debugging. For instance, SoftICE, a kernel mode software debugger, contains a well-known backdoor that allows debugged processes to communicate with it. As a result, a process can check for SoftICE's presence by attempting to access the backdoor communication channel.[3]

Processes can also detect human behavior. When a process detects an unexpected pause in execution, for

example, it can presume that a human paused the process by using a debugger. It's trivial for processes to detect pauses in execution; they simply take two time samples and compare them—a large difference between two time samples implies that a human has paused the process. Processes can take samples from multiple sources including hardware, the operating system, and external computers in a network.

Penalizing the debugger

By itself, detection doesn't thwart debugging: following detection, a debugged process must penalize the debugger. Processes can impose many potential penalties on debuggers, but the simplest one is termination. Once a process becomes aware of debugging, it can simply decide to exit. Although this might seem like the most obvious action following detection, it isn't ideal—not only does immediate termination tip off the reverse engineer that the process has detected the debugging attempt, it also lets him or her know the anti-debugging code's location. At this point, the analyst will likely restart the process and attempt to circumvent the anti-debugging trick.

The best penalty to choose depends on the purpose of protection. If a process is trying to protect an algorithm's intellectual property, for instance, it could surreptitiously avoid the algorithm, execute a more conventional algorithm when debugged, covertly cause the algorithm to behave abnormally, or introduce a fault that causes itself to crash later during execution.

Detection is only useful if the process chooses its penalties wisely. Remember, the end goal of anti-debugging is to thwart the entire debugging system.

Exploitation

Processes can attack debuggers by exploiting any of the specific debugging subsystems. To do so, it must first have a vulnerability, which is often the result of intended features, unintended bugs, or weaknesses in the debugging subsystems.

The Intel x86 architecture, for example, specifies that hardware breakpoints are ineffective when placed on instructions immediately following a `POP SS` or `MOV SS` instruction.[4] This is an intended feature of the architecture—it prevents the existence of invalid stacks during interrupt calls. Figure 2 shows how this design feature can be used to shield instructions from hardware debugging.

The user-mode software debugger, OllyDbg, contains a bug that lets processes ignore breakpoints placed on their entry points. Figure 3 shows an example of modifying a program's Portable Executable (PE) header to thwart debugging. At load-time, OllyDbg interprets these values as evidence of an invalid binary image. Although OllyDbg considers the program to be invalid, it still allows it to be executed. When executed, any breakpoints placed on the process's entry point will be ignored, allowing those instructions to be executed without being debugged.[5]

Processes can also target human weaknesses for exploitation in several ways. Because debugging is only successful when the reverse engineer gains useful knowledge, taxing the reverse engineer increases the amount of time that must be spent debugging a process. Thus, a process that makes debugging extremely arduous increases the likelihood that the reverse engineer will eventually give up. Processes can do this by overloading reverse engineers with meaningless information such as red herring functions and data, or they can confuse reverse engineers by burying their algorithms in complex obfuscations. A reverse engineer's patience is ultimately exploited through the sum of all the anti-debugging techniques a process can present.

Overcoming anti-debugging

We've discussed a variety of ways in which processes can thwart debuggers: penalizing the debugger following passive and active detection; exploiting features and bugs in the debugger; and targeting inherent human weaknesses to confuse, annoy, and perplex the person sitting behind the keyboard. At the end of the day, though, how effective are attacks on debuggers? Can they always stop reverse engineering? What are the limitations of these attacks? How can reverse engineers overcome these attacks?

There are a few of points to make here. First, every debugging attack targets a specific debugging subsys-

```
mov eax, dr7
cmp eax, 400h
jnz Hardware_BPX_detected
```

Figure 1. Passively detecting a hardware debugging subsystem. The debug register `dr7` will contain the value `400h` if it's unused.

```
PUSH SS
POP SS
CALL secret_function
```

Figure 2. Exploiting a feature of the Intel x86 hardware debugging subsystem. Instructions can be executed covertly by calling them after executing `POP SS`. Hardware breakpoints can't be set after the `POP SS` instruction is called, so the secret function instructions are shielded from debugging.

```
Portable Executable (PE) Header:
...
LoaderFlags:  0XABDBFFDE
NumberOfRvaAndSizes: 0XDFFFDDDE
```

Figure 3. Exploiting an OllyDbg bug. Based on these invalid flags, OllyDbg will believe the program is invalid, but still allow program execution to continue. However, any breakpoints set on program entry will be ignored, permitting instructions in program entry to be executed without being debugged.

tem. One attack might target SoftICE, for example, so if the reverse engineer isn't using SoftICE, then that attack will fail. Debugging subsystems that aren't specifically targeted by a process are impervious to attack. Processes usually assume—incorrectly—that some debugging subsystems are always used. When a program is compiled for the Intel x86 architecture, for instance, it seems reasonable to assume that attacks on the Intel x86 debugging subsystem will be effective. However, an emulator might mimic the hardware layer, thwarting attacks against the hardware debugging subsystem. Though bear in mind that emulators can be attacked, too.

The second main point regarding the limitations of attacks on debuggers is that reverse engineers can patch anti-debugging techniques. If they realize their debuggers are under attack, reverse engineers will attempt to circumvent that attack. Processes can prevent some circumvention efforts through tamper-proofing, but by itself, tamper-proofing isn't perfect—processes can't prevent tampering with 100 percent certainty.

Stealth debuggers

The best way to prevent a reverse engineer from circumventing an attack is to execute the attack covertly. If analysts don't realize they're being attacked, they'll never attempt to circumvent the attacks.

Likewise, stealth debuggers represent a significant threat to anti-debugging techniques: they're impervious to attack because what the attacker doesn't see can't be exploited (ideal stealth debuggers are completely undetectable and unexploitable). They can detect specific debugging detection methods and return falsified results to processes and they can be designed to minimize the possibility of containing exploitable bugs. Although ideal debuggers don't exist today, researchers are working to create them.[6]

Although stealth debuggers can circumvent most anti-debugging tricks in theory, some attacks exist that stealth debuggers can't overcome automatically—timingchecks and human-layer attacks, for example. Some detection methods based on timing checks are very difficult to automatically circumvent. Granted, it would be trivial for a stealth debugger to intercept hardware and operating system time queries; however, stealth debuggers will not be able to intercept and return all external time queries with 100 percent accuracy because a process can query the time through a network in an infinite number of ways. Also, stealth debuggers can't automatically prevent attacks on the human layer because we'll always have exploitable weaknesses.

Attacking the debugger can be an effective way to thwart the reverse-engineering process. At the end of the day, though, who wins? Can software developers use anti-debugging to protect their software? Will security researchers be able to overcome the anti-debugging tricks embedded in the latest-and-greatest malicious software sample?

On one hand, talented and patient reverse engineers can manually circumvent most anti-debugging tricks. Stealth debuggers, once developed, will also give reverse engineers a significant advantage. But on the other hand, covert anti-debugging methods provide some software protection because reverse engineers can't manually circumvent anti-debugging techniques they don't see. By carefully penalizing the reverse engineer, protected software stands a chance, even in the hands of experienced reverse engineers.

Currently, there are enough anti-debugging techniques available to software engineers to sufficiently protect software against most threats. Likewise, most state-of-the-art malware can be sufficiently reverse-engineered with patience and skill to enable security researchers to continue to defend their networks. However, advances in software protection techniques and reverse engineering might alter the balance. □

References

1. P.C. van Oorschot, "Revisiting Software Protection," *Proc. 6th Int'l Conf. Information Security* (ISC 03), LNCS 2851, Springer-Verlag, 2003, pp. 1–13; www.scs.carleton.ca/~paulv/papers/isc5.pdf.
2. K. Kaspersky, *Hacker Disassembling Uncovered*, A-List Publishing, 2003.
3. P. Cerven, *Crackproof Your Software: Protect Your Software Against Crackers*, No Starch Press, 2002.
4. Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3: System Programming Guide*, 2006; www.intel.com/products/processor/manuals/index.htm.
5. N. Brulez, "Scan of the Month 33: Anti Reverse Engineering Uncovered," 2004; www.honeynet.org/scans/scan33/nico/index.html.
6. A. Vasudevan and R. Yerraballi, "Cobra: Fine-Grained Malware Analysis using Stealth Localized-Executions," *Proc. 2006 IEEE Symp. Security and Privacy* (SP 06), IEEE CS Press, 2006, pp. 264–279.

Michael N. Gagnon *is a research staff member in the Center for Secure Information Systems at George Mason University. His primary research interest is software protection. Gagnon has a BS in computer science from George Mason University. Contact him at mgagnon1@gmu.edu.*

Stephen Taylor *is a research staff member in the Center for Secure Information Systems at George Mason University. His research interests include software protection, computational fluid dynamics, and gaming. Taylor is pursuing his BS in computer science at George Mason University. Contact him at staylor5@gmu.edu.*

Anup K. Ghosh *is chief scientist in the Center for Secure Information Systems at George Mason University. His research interests include malicious code analysis, virtualization for security, and systems security. Ghosh has a PhD and an MS in electrical engineering from the University of Virginia. He is a senior member of the IEEE. Contact him at aghosh1@gmu.edu.*