A Highly Immersive Approach to Teaching Reverse Engineering

Golden G. Richard III
Department of Computer Science
University of New Orleans
New Orleans, LA 70148
Email: golden@cs.uno.edu

Abstract

While short training courses in reverse engineering are frequently offered at meetings like Blackhat and through training organizations such as SANS, there are virtually no reverse engineering courses offered in academia. This paper discusses possible reasons for this situation, emphasizes the importance of teaching reverse engineering (and applied computer security education in general), and presents the overall design of a semester-long course in reverse engineering malware, recently offered by the author at the University of New Orleans.

1 Introduction

Reverse engineering of software involves detailed analysis of the low-level structure and run-time effects of a software component. The component under investigation might be an entire application, a kernel module, a software patch, or a single function. Reverse engineering is often applied directly to application binaries (or bytecode, in the case of applications written in interpreted languages), in the absence of available source code. Reverse engineering has many uses, including facilitating software interoperability, evaluations of the effects of security patches, security auditing (e.g., determining if the effects of an application are malicious or unwanted), and enhancing software functionality or performance. It might also be performed out of simple curiosity, to understand how a software component works, or to "crack" software to defeat copy protection techniques. Reverse engineering also plays a critical role in the deep understanding and attempted mitigation of malicious software and this paper focuses on that role.

The skills needed for effective reverse engineering are diverse and some are not emphasized in current-generation computing curricula. Reverse engineering requires not only tenacity (a primary skill), but also strong assembler language skills, knowledge of operating systems internals and both low-level and high-level APIs, and in many cases, significant knowledge of hardware details. An abstract view of how paging works or a brief description of what a translation lookaside buffer (TLB) does is considered sufficient in most operating systems courses, but for reverse engineering, the devil really is in the details. For example, to understand an offensive technique like Shadow Walker [13], which relies on de-synchronization of the data and instruction TLBs in Intel's split-TLB design to hide code and data, it's not enough to have seen a Powerpoint slide depicting the abstract functionality of a TLB—that's a good start, but both more details and hands-on experience are needed. Furthermore, assembler language courses, if they exist at all as independent courses in a modern computing curriculum, tend to be much weaker than in the past, often emphasizing the use of High Level Assembler (HLA) [4] and development of toy applications. In many curricula, the assembler language course of old has been folded into the undergraduate architecture class. This is understandable to some degree because of the limited use of assembler in general-purpose computing, but is a challenging issue facing someone developing a reverse engineering course.

When the author began development of the reverse engineering course described in this paper, he performed a routine search for other reverse engineering courses—what are others doing? What tools are they using? How are they managing to address all of the necessary skills in a single semester? The search yielded little. There appeared to be almost no reverse engineering courses being taught at all in academia. Why? Student disinterest can't be the reason—the author's own students were wildly enthusiastic, despite dire warnings of severe mental pain, lost sleep, and nightmares involving segmentation and instruction prefetch caches. Why are there almost no hardcore reverse engineering courses in academia? One possibility is that there is a perception that necessary skills can't be developed in a single semester. While only additional experience beyond a single course can fully develop a student's reverse engineering skills, an impressive set of skills can certainly be fostered within a single semester. Another possibility is the fear (on the part of either faculty members or administration) that teaching reverse engineering isn't a good idea. What impact will such a course have on the school's computing infrastructure? Should we be teaching students to

"crack" software? Precautions are necessary, but avoiding horror stories isn't difficult, given a modest amount of dedicated computing resources. A third possibility is that a reverse engineering course might not fit into curricula that increasingly emphasize "abstract", hands-off computing. Finally, developing a reverse engineering course, at least for the first time, requires a *significant* amount of effort for the faculty member—it's far easier for faculty members to flip Powerpoint slides than to prepare and then grind through long assembly listings with their students. Whatever the reason, hands-on, applied computing is *essential* if we are to properly develop systems-oriented computer scientists who are able to solve important problems in computer security.

The thesis of this paper is that hardcore reverse engineering can be taught effectively, with a non-fatal degree of effort on the part of a dedicated instructor, and with laboratory facilities that are within the reach of most departments. Furthermore, the effort is well worth it, and trains students not only to "do" reverse engineering, but also to be much better systems people, providing them with better grounding in architecture, assembler language, operating systems internals, and software optimization, all as a result of their hands-on experiences. Studying reverse engineering teaches students how to look hard at the *details* of systems, rather than just noting whether systems do or do not have some desired effect.

The remainder of the paper describes the laboratory setup, topics covered in the reverse engineering course, teaching methods, and a discussion of the particular malware samples analyzed by students in a reverse engineering course taught by the author.

2 Course Details

2.1 What is it, Really?

The course described in this paper isn't a traditional computer security course, nor even a traditional academic course, in the "listen to lectures and do a few laboratory exercises" sense. Neither is it a hacking course, although the topics covered could be watered down sufficiently to fit within the scope of a single semester course on hacking or offensive computing. Instead, the goal of the course is to teach students, within a single semester, the necessary skills to examine and understand malicious software.

2.2 Laboratory Setup

The Department of Computer Science at the University of New Orleans has a dedicated laboratory for digital forensics research and instruction, which was used for teaching the reverse engineering course. This laboratory contains high-end workstations that boot Linux and run Windows XP (as well as a variety of other guest operating systems) under VMWare. The lab is served by a dedicated gigabit network switch that can be isolated from the main departmental network and a dedicated file server. Linux boots from the drives installed in the workstations, but all user files, including the VMWare images for Windows XP on the workstation, are stored on the fileserver. The reverse engineering course concentrated primarily on reversing Windows binaries, so virtually all analysis was performed under XP running as a guest inside VMWare. Students were instructed to turn networking off inside VMWare during analysis of malware and to make heavy use of snapshots to maintain the integrity of their analysis environments. While "jump of out VM" proofs of concept now exist that allow arbitrary code execution on the host OS from a guest OS, no software of this type was analyzed in the reverse engineering course. Furthermore, while the students were taught essential anti-VM techniques and various workarounds, the majority of the malware that they analyzed did not actively use anti-VM measures, to increase the usefulness of the VM-based analysis environment.

The use of VMWare allowed us to push a single user environment for reverse engineering, packaged in a 20GB VMWare image, into each student's account. This environment could then be easily refreshed if contaminated during malware analysis. The environment for malware analysis contained the following software:

- The sysinternals tools [15], including procmon, regmon, etc. for dynamic analysis. These tools are freely downloadable.
- Visual C++ Express Edition [7], which is available free to students. Each student activated her own copy of Visual C++ inside her private VMWare image. While the university has a site license for the standard Visual C++ distribution, the use of Express Edition was ideal, because it allowed students to migrate the package to their personal machines for work outside the laboratory.
- The MASM32 SDK [6], which is freely downloadable and contains necessary include files and libraries for writing assembler using Windows operating systems.
- OllyDbg [9], a very popular Windows debugger, which is freely downloadable.
- IDA Pro [5], a disassembler with extensive support for plugins, for static analysis. This was the only essential commercial product. For most of the

analyses performed in an introductory reverse engineering course, even the freeware 4.x version is sufficient, if resources don't permit purchase of sufficient 5.x licenses. The most important plugin for IDA Pro in the context of the course was x86emu [16], which provides limited x86 emulation capabilities within IDA Pro (useful, e.g., for simple unpacking operations before a static analysis is performed).

- HBGary's Responder [3], useful for dynamic analysis of malware. This product is relatively expensive, but HBGary was kind enough to offer licenses for use by students in the UNO laboratory.
- A number of research-grade tools for live forensics analysis [2][10][12], to emphasize the important connections between traditional digital forensics analysis and malware analysis / incident response.

2.3 Topics Covered

The course covered a number of topics in detail. The approach used to manage the introduction and mastery of these topics in a single semester is covered in Section 2.4.

The list of topics included:

- Goals of reverse engineering, static and dynamic analysis, limits of each type of analysis.
- Ethics and legal issues, including the DMCA and the impact of EULAs on reverse engineering efforts. While the analysis of malware rather than commercial software alleviates most legal issues, the need to seek legal counsel before pursuing most reverse engineering projects was stressed heavily.
- Description of available tools for reverse engineering, including disassemblers, debuggers, live forensics tools, etc.
- Types of malware, typical propagation and payload delivery strategies, poly- and metamorphic malware.
- Basic Intel assembler, including information about registers, flags, common instructions, instruction and data formats, differences between 32-bit and 64-bit code, interaction with important hardware components, including the paging and debugging architectures.
- Basic DOS and Windows internals and the most important APIs for malware analysis. Because of the breadth of this topic, much of this had to be learned "on the job" by students, during their malware analysis assignments.
- Windows Portable Executable (PE) format. Deep understanding of executable file formats is crucial for analyzing modern malware, since malware often parses and modifies executable files to hide and to propagate. PE format was stressed in the course because the emphasis was on malware that impacts Microsoft Windows.
- Typical control structure, function, array, and C struct/union patterns used by commonly encountered compilers when compiling C into assembler.
- Common malware functionality, including delta offset calculation**,** discovery of entry points for needed APIs, infection and propagation, etc.
- A catalog of anti-disassembly, anti-VM and anti-debugger techniques, including the use of self-modifying code, dynamic jumps, instruction prefetch attacks, local and global descriptor table location analysis, and Windows APIs for debugger detection.
- Packing and unpacking techniques, internals of popular packers such as UPX [14] and Armadillo [1]. Small, simple programs were packed and unpacked to illustrate the necessary steps, before malware was considered.

2.4 Approach

Teaching reverse engineering to students who possess a basic systems background, but who largely lack adequate assembler or OS internals skills, is a challenging task, given a typical 15 week semester. The solution for rapidly developing necessary skills is to teach the basics of reverse engineering while continuously immersing students in malware analysis, from the very first lecture. By starting with analyses of malware samples that don't require extensive knowledge of, e.g., Windows APIs, and interleaving analysis with traditional lectures on topics relevant to near-future assignments, it *is* possible for students to develop impressive skills within a single semester.

A variety of teaching strategies and assignment types were used to cover the necessary material while developing the reverse engineering skills of the students. These included:

- Team-oriented analysis of malware samples. All assignments were performed in teams, with active team participation checked via traditional examinations (see below). The typical size of a team was 2, with teams of 3 students occasionally approved by the instructor.

Refer to the following partial disassembly of the Harulf virus. Neatly insert comments for each line that clearly explain what the instructions / directives accomplish.

```
Start:
          jmp stuck
          sig_1 dd 0
          sig_2 dd 0
stuck:
          call here
          jmp getdelta
here:
          assume fs:nothing
          mov eax,[esp]
          push eax
          push fs:[0]
          mov fs:[0],esp
          xor eax,eax
          mov eax,[eax]
          ret
getdelta:
          ...
          pop fs:[0]
          pop edx
          pop ebp
          sub ebp,offset here
          add ebp,2h
          cmp ebp,0
          je skipdecrypt
```

Figure 1. A sample midterm examination question that tested student participation in team-oriented reverse engineering exercises.

- Traditional lectures, with Powerpoint slides, to introduce important background material, as described in Section 2.3.
- Source code walkthroughs, with a document camera connected to an LCD projector, for a number of malware samples, including every sample analyzed by the student teams. For these walkthroughs, fully-documented assembler was used (documentation provided by the instructor) and further marked up during class discussion. The marked up copies were then scanned and distributed electronically to students. These pencil and paper sessions with a document camera were essential to the success of the course, for several reasons. First, the document camera allows large portions of the malware sample to be viewed at once, which is simply not possible with Powerpoint. The in-class markup also brings persistent context to the discussion and provides a reasonable pace for the presentation.
- Traditional midterm and final examinations, with the bulk of questions directly related to the malware analyses conducted by the teams. For example, a typical question might target a tricky section of the disassembly of a familiar malware sample. If the student actively worked on the analysis, then the question should be answerable within the allowed time. The question will pose significant difficulty if the student didn't participate in the malware analysis. Figure 1 provides an example, based on team-based analysis of the Windows Harulf virus. Answering the question correctly requires analyzing Harulf's delta offset calculation as well as one of its anti-debugging strategies (involving structured exception handling combined with a null pointer reference to trip up debuggers). Questions like this are sufficiently difficult that reasonable reverse engineering skills are required to answer them in case the student didn't actually work on the analysis with her team. If the student can answer questions of this kind without participating in the analyses, then in the author's opinion, the student still meets the course's goals.
- Short, in-class, laboratory sessions. In addition to out-of-class team assignments, a number of in-class laboratory sessions were conducted, to allow the instructor to directly assist students with new

analysis techniques, such as unpacking executables. These sessions were graded as pass/fail, with attendance and completion of an assigned task resulting in a pass.

The next section discusses representative malware samples analyzed in class and used for team assignments. Presentation of important topics using traditional lecture mechanisms was interleaved with in-class analysis of these malware samples (and others) to motivate students to study necessary background material. For example, the Michelangelo virus was analyzed during the first week of class, along with handouts describing essential DOS internals necessary to fully understand the actions taken by the virus.

2.5 A Malware Sampler

This section discusses some of the primary malware samples used in the reverse engineering class, in order of presentation, with a brief justification for each. There is nothing magical about these selections other than the fact that the sequence of malware samples meets two main requirements. First, the sequence should progressively introduce new and important features that a reverse engineer might encounter when analyzing modern malware. Second, malware samples in the sequence should pose a reasonable increase in difficulty, sufficient to repeatedly challenge student teams nearly to the limits of their current ability (within available time).

Michelangelo

Michelangelo is a DOS boot sector virus that is transmitted by infecting the boot sector of floppies and the MBR of hard drives. It stays resident to infect additional floppies by replacing the interrupt handler for floppy I/O under MS-DOS. On Michelangelo's birthday, the virus maliciously overwrites sectors on attached hard drives. Michelangelo is a good choice for a gentle introduction to reverse engineering because it yields easily to static analysis. It introduces students to boot sector and master boot record (MBR) layouts, simple destructive payloads, and provides a good introduction to many DOS system calls, all of which are executed in a straightforward manner. It is interesting enough to hold the attention of students and difficult enough to motivate students to study necessary background materials and work on improving their assembler skills.

DOS-7

The DOS-7 virus infects .COM files, including a specific version of COMMAND.COM. The virus employs a number of mechanisms to conceal its intent and prevent analysis. One example is obfuscating systems calls for file I/O by remapping the *int 21h* interrupt handler. Another is employing self-modifying code to hide the values of arguments to system calls, which requires hand disassembly to unravel. The virus also employs a simple but deadly (at least on older Intel processors) anti-debugger trick. When analyzed inside a debugger on older Intel hardware, differences in instruction prefect cache execution under DEBUG.EXE are exploited to partially erase attached hard drives.

SQL Slammer

SQL Slammer is a UDP-based, single packet worm that attacks Microsoft SQL Server installations. The rapid propagation of SQL Slammer caused extensive damage in 2003 [8], even though the worm carries no malicious payload. SQL Slammer was used as the first Win32 malware sample because it is reasonably easy to understand, employing only limited obfuscation and reliance on a small set of Windows APIs.

Lucius

Lucius is a Windows virus that recursively visits directories to infect .EXE files and infects employs a single level of XOR-based encryption, introducing students to simple packing strategies. Lucius has no malicious payload, uses kernel32.dll detection that works under both Win9x and WinNT+, and patches CALL statements in infected EXEs to divert execution to the virus' code.

Harulf

Harulf is a flawed Windows virus that contains bugs that impact its ability to deliver its multiple payloads, but is still interesting enough to assign for analysis, for a number of reasons. It employs simple polymorphism, two layers of encryption, as well as a number of anti-debugging strategies, including the use of a structured exception handler within the delta offset calculation and detection of anomalous execution of CPUID instructions. Much of Harulf is copied from other viruses (including its code for detection of the location of kernel32.dll, critical for access to other needed APIs), which allows students to use online resources to crack some of its secrets.

Conficker

The class did not analyze a Conficker sample, though a sample was available. Instead, the class reviewed the SRI analysis of Conficker C [11] as a group, during the last week of the semester. Conficker employs multiple defenses against analysis and has novel, secure dynamic update mechanisms. The use of the Conficker analysis to "shut down" the semester provided students with a chance to understand the serious difficulties in

analyzing state-of-the-art malware and to reflect upon their current skill levels.

3 Discussion

The reverse engineering class discussed in this paper was taught in Spring 2009 to 25 students, approximately 2/3 of whom were graduate students, with the remainder being undergraduates. The students entered the class with a wide variety of backgrounds and technical competencies. All of the students had taken an introductory course in computer security, which discussed malicious software at a high-level, but did not expose them to serious reverse engineering efforts. Approximately 20% of the students had previously taken an operating systems internals course (concentrating on Linux and also taught by the author). Approximately 50% of the students had taken or were simultaneously enrolled in one of the courses in our digital forensics curriculum. Perhaps most important, virtually none of the students had taken a serious assembler language course or had expert-level assembler skills.

Without a doubt, the students that enroll in a reverse engineering course tend to be "hacker" types, interested in highly applied and experimental computing. Still, there were non-isolated cases of trepidation (some severe) among the students when they discovered that the course would be both very immersive and in most cases, seriously tax their limited assembler language skills. Only one student dropped this course in Spring 2009 and while several students remarked that it was the hardest course they had ever taken, none failed. The skill levels of students upon completion varied considerably, from average to extremely competent. Several students enquired seriously about a "Reverse Engineering II" course—no such course is planned and in the author's opinion isn't necessary.

4 Conclusions

This paper discusses a reverse engineering course recently developed by the author and taught at the University of New Orleans. The course uses a highly immersive approach, continuously exposing students to increasingly more difficult malware analysis, both in class and with team assignments, while interleaving traditional lectures on topics such as advanced assembler, Windows APIs, and packing. A crucial pedagogical element of the course is the use of detailed, in-class walkthroughs of malware source code using a document camera. The source code is thoroughly marked up based on class discussion and scanned copies of the commented and marked up source code are then distributed to students electronically. Student response to the course was overwhelmingly positive. In the author's opinion, as well as that of many of his colleagues, courses such as the one described, which concentrate on applied, hands-on security analysis, are absolutely necessary if academia is to increase production of students capable of solving real-world security problems.

Acknowledgements

The author would like to thank Andrew Case and Lodovico Marziale, who contributed to the development of the course. Thanks to Vitaly Shmatikov, Somesh Jha, and Eugene Spafford for discussions about the state of reverse engineering in academia. Finally, thanks to the anonymous referees for comments that improved the final version of the paper.

References

[1] Armadillo packer, *http://www.siliconrealms.com*.

[2] A. Case, A. Cristina, L. Marziale, G. G. Richard III, V. Roussev, "FACE: Automated Digital Evidence Discovery and Correlation," *Proceedings of the 8th Annual Digital Forensics Research Workshop (DFRWS 2008)*, Baltimore, MD, 2008.

[3] HBGary Responder, *http://www.hbgary.com/*.

[4] R. Hyde, "High Level Assembler *Language"*, http://webster *.cs.ucr.edu/AsmTools/HLA/index.html.*

[5] IDA Pro, *http://www.hex-rays.com/idapro/*.

[6] MASM32 SDK, *http://www.masm32.com/*.

[7] Microsoft Visual C++ Express Edition, *http://www.microsoft.com/Express/vc/*.

[8] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, N. Weaver, "Inside the Slammer Worm," *IEEE Security and Privacy*, 1(4):33-39, July 2003.

[9] OllyDbg, *http://www.ollydbg.de/*.

[10] N. Petroni, A. Walters, T. Fraser, and W. Arbaugh, "FATKit: A Framework for the Extraction and Analysis of Digital Forensic Data from Volatile System Memory," *Digital Investigation*, 3(4):197-210, December, 2006.

[11] P. Porras, H. Saidi, V. Yegneswaran, "Conficker C Analysis", *http://mtc.sri.com/Conficker/*.

[12] A. Schuster, "Searching for Processes and Threads in Microsoft Windows Memory Dumps", *Proceedings of the 2006 Digital Forensic Research Workshop (DFRWS)*, 2006.

[13] S. Sparks, J. Butler, "Raising the Bar for Windows Rootkit Detection," *Phrack* Issue # 63.

[14] "UPX: The Ultimate Packer for eXecutables," *http://upx.sourceforge.net/*.

[15] Windows sysinternals suite, *http://technet .microsoft.com/en-s/sysinternals/default.aspx*.

[16] ida-x86emu plugin for IDA Pro, *http://ida-x86emu.sourceforge.net/*