Code Hardening: Development of a Reverse Software Engineering Project

Cynthia C. Fry, Zachary Steudel
Department of Computer Science
Baylor University

Abstract

In CSI 2334, "Introduction to Computer Systems," we introduce a group project to the students whose purpose is to simulate a team project on the job. Group projects are used very frequently to provide a similar learning environment which capitalizes on the benefits of peer-to-peer instruction, or cooperative learning. In this group project, students are presented with a challenge. A piece of executable code has been found on an older server, and the student teams must determine what the code is designed to do; and, in particular, whether the code is benign or malicious in nature.

In order to simulate this scenario a software system is developed and then hardened. The system developed is generally a game that has some malicious content, which is then obfuscated before the executable is presented to the student teams. The objective for the student teams is to research methods by which the binary file can be "read" without executing it, and to modify the behavior of the executable file, depending on the purpose of the code.

This paper will introduce methods of software hardening and obfuscation; and will discuss the design of the project, the implementation of the design, code obfuscation techniques used, and which obfuscation techniques were used to produce the mystery executable presented to the class as their class project.

Introduction

Before introducing the methods and challenges related to hardening and obfuscating a code executable for a classroom project, it is important to understand the depth of knowledge available in the field and the maturity of the discipline at this point.

Code-level hardening changes can be defined as "changes in the source code in a way that prevents vulnerabilities without altering the design"[1]. Code hardening is a common practice in major software development firms, with many teams carving out sprints[2] to harden new code by refactoring and removing unnecessary dependencies[3]. Code-level hardening is also employed to make it difficult for attackers from easily discovering vulnerabilities in the code, as well as to protect the program owner's intellectual property[4].

---

[1] (Mourad, Debbabi, & Laverdière, 2006)
[2] A sprint is generally defined as a short 2-4 week iteration in a team's software development process. Most agile development processes utilize sprints to develop software.
[3] (Scarlett, 2017)
[4] (Udupa, Debray, Madou, 2005

Hardening can also refer to practices that make the original source code of an application more difficult to understand. This is generally defined as code obfuscation. The practice of code obfuscation in modern software engineering dates back to the 1980's, with small competitions held to transform simple C code into confusing, abstract puzzles difficult for humans to parse and understand[5].

Since the 1980's, literature on code-level obfuscation has been consistent but generally sparse. This field of research is relatively small, with no more than three or four papers published each year since the early 1990's. With the growth of cloud-computing services that require paid subscriptions and API keys embedded in the code, the necessity of obfuscation techniques to halt reverse software engineering attempts has grown. Without sufficient code obfuscation, reverse engineers can easily disassemble or decompile code to gain access to API keys and compromise an application's access to cloud databases or cloud compute resources[6].

Introduction to the Challenge

Students in CSI 2334, "Introduction to Computer Systems," are paired up for the class group project a little more than halfway through the semester. Each pair of students is presented with a Windows executable file with no information other than that this file has unknown origins and unknown functionality. Given this, students are tasked with learning the effects and purpose of the executable and to quarantine any malicious elements while also further developing the functionality of the program.

The CSI 2334 course is primarily focused around low-level design, architecture, and code. This project plays into the goals of the course by mimicking the research and design process of small agile teams in the software engineering industry while also developing the low-level computing skills of the students. Given only an executable file, students must take advantage of open source software to gain access to the binary code and modify it. At the end of the semester, the student groups present their findings to the professor and the teaching assistant in a formal mock business setting.

Determination of the Challenge

Before developing the project for CSI 2334, it was first necessary to analyze the learning objectives of the project and potential outcomes. Students are strongly encouraged to develop their collaboration and research skills throughout the extent of the challenge. Given this, it was important to design a program that included some layers of complexity that prompt students to begin researching. It was decided early in the design process to write the project's program in the C programming language, adding one layer of complexity to the project, as students up to this point have mostly worked with the C++ language. Although these languages are similar, students' unfamiliarity with some of the lower-level conventions of C already provides one small layer of

[5] (Xu, Zhou, Kang, & Lyu, 2017)
[6] (Code hardening, 2019)

obfuscation to the executable.

After determining the language to write the program, it was critical to determine the functionality of the project executable. It was quickly decided to make the program into a simple game. Given that students are tasked to make changes to the executable that they are given, a game provides many chances to make clever edits. With a game, students can add cheat codes, game modes, and even change base-level game functionality in interesting and unique ways. After deliberating over the game to create, it was decided to make Snake in C. Snake is simple; players travel a game board with the arrow keys to eat food and grow larger. The game ends when the player hits a wall or hits their own body. Their goal is to eat as much food as possible before colliding with something, as food adds to the player's score. Making the project Snake provided many opportunities for changes in the binary code, as students can change how the score is calculated, how losing works, how food is spawned, and much more.

With the core functionality of the project finalized as a Snake game written in C, it was important to add a few more layers of complexity to push students to stretch their research and collaboration skills further. It was decided to wrap the game with a malicious fork bomb to prevent students from simply running the executable upon retrieving the project. A fork bomb is categorized as a denial of service attack in which "the user repeatedly forks new processes," absorbing computer resources "mak[ing] the system unusable for others"[7]. To create a fork bomb in a Windows executable created in C, the system() function was utilized in a for-loop. With each call to the system() function, a command was sent to open ten terminals on the user's computer. Within seconds of running this, there are hundreds of terminals open on the user's computer each utilizing system resources and tremendously slowing down new inputs.

To avoid potentially damaging the computers of students in CSI 2334, the fork bomb was determined to be finite, ending after 1,000 calls to the system() function, opening a total of 10,000 terminals on the user's computer. Users initiate the fork bomb within the Snake game by pressing a key on their keyboard. Most students would find this malicious element if they ran the game and tried to move the snake on the game board. The purpose of this malicious layer in an otherwise normal game is to spur more research on tools and techniques to mitigate the issue. The fork bomb can potentially crash a user's computer, so with this threat looming, students will need to discover virtual machine tools to run the executable.

At this point in the determination of the challenge, it has been decided that the project is to be a Snake game written in C that is wrapped in a malicious computer-crashing fork bomb. Multiple layers of complexity exist at this point to spur group research and collaboration, however with the rapid iteration and development of modern reverse engineering tools such as Ghidra and IDA, it was necessary to add one more layer to the challenge. If the executable was to be given to students in its current state, it would only take minutes after loading the file into a disassembler such as IDA to locate the fork bomb and neutralize it. This is where obfuscation techniques such as junk code insertion, rabbit-hole function calls, and stack pointer manipulation have been used. With these obfuscation techniques, students are required to do much more complex analysis in the Assembly

[7] (Berlot & Sang, 2008)

and hex code of the executable to determine what code is necessary.

Design, Testing, Obfuscation

With the general layout of the project program finalized, it was important to plan out the implementation process for the game. It was decided to first implement the core game functionality. To verify that core game functionality is correct, multiple exhaustive rounds of the new Snake game were to be played. After ensuring that the base functionality of the game was complete and correct, the hardening and obfuscation was to be included.

As previously mentioned, obfuscation techniques were to be junk code insertion, rabbit-hole function calls, and stack pointer manipulation. Junk code insertion is a common and relatively simple technique in basic code hardening and obfuscation. The goal of this technique is to muddy the source code of an application without changing the functionality of the application. To do this, random variables are allocated and never used and function calls that allocate memory are called upon, with none of these affecting the main gameplay. These insertions are placed randomly throughout the source code, with the goal of prompting students to analyze the source code more closely to decipher functionality of each section of code.

Similar to junk code insertion, rabbit-hole function calls do not affect the gameplay of the program but instead serve to confuse the reverse engineer by de-linearizing the execution flow. Mixed in with the necessary program code for the Snake game, there are rabbit-hole functions that are called that make further function calls that serve no purpose for the game itself.

Finally, stack pointer manipulation is another obfuscation technique used to muddy the execution flow of an otherwise normal program. By using inline Assembly instructions within the C code, the stack pointer can be directly manipulated. This technique is key to challenge reverse engineers using a graph execution view of a program, as stack pointer manipulation will throw off the execution graph. This obfuscation technique further serves to force students to research other techniques and technologies in their reverse engineering journey, as they cannot simply rely on one method of reverse engineering for the whole project.

Agile Development of the Challenge

To develop multiple layers of complexity into the Snake game project, it was important to tackle the program with a plan. When developing the program, a test-driven process was used. With each feature developed within the program, time was spent testing and verifying the correctness of the program before moving on to the next feature. Before adding any hardening or obfuscation, the main Snake game was completed and tested for correctness.

Once it was verified that the base Snake game was correctly working, the hardening and obfuscation began. With each layer of program obfuscation, short manual tests were run to verify that the program functionality had not been changed. The first layer of obfuscation added was junk code insertion. Snippets of memory allocation and useless function calls were injected into the main game

code and then tested to ensure no tampering with the game functionality had occurred. After junk code insertion, rabbit-hole function calls were similarly inserted into the main game code. Again, it was subsequently tested that the rabbit-hole function calls caused no effect within the game itself. Finally, all variable and function names were changed to arbitrary garbage names so as to add one final layer of obfuscation.

With the program obfuscation complete, the malicious element of the game was to be added. Within the main game control loop, a function call was added within the if statement that checks whether the user entered a key or not. The called function contains the code to open 10,000 terminals on the user's computer. To test and verify this feature, the amount of terminals produced by the fork bomb was reduced to 50. Once verified that the fork bomb created the expected number of terminals upon triggering, the number was changed back to 10,000 and the program executable was exported.

Summary and Conclusions

Upon finishing the semester and listening to the final presentations of the students, a retrospective look was taken on the project. There were a few key takeaways from this retrospective.

- Reverse software engineering tools have become advanced enough that just a few layers of program obfuscation do not provide a sufficient challenge. With the release of the open-source NSA software Ghidra in April of 2019, reverse engineering has been simplified. Ghidra packages all the typical reverse engineering tools such as disassemblers, decompilers, binary editors, and more, allowing many students to rely on Ghidra alone to finish the class project. To encourage the use of more tools and more research, further layers of obfuscation that take advantage of the weaknesses of Ghidra will need to be developed into future class projects.
- Malicious elements of future projects should be deeper and more complex. Using Ghidra and other modern reverse software engineering tools, students were easily able to find the fork bomb within the Snake game and quarantine it with simple binary code editing. In future iterations of the project, multiple malicious elements alongside obfuscation of those malicious elements will be necessary to provide a greater challenge to students.

Other than these takeaways, it was clear that this project provided a great outlet for students to strengthen their research and collaboration skills. Given that most students entered the 2334 course with no knowledge of reverse software engineering, the project successfully simulated a short professional software engineering project that required fast learning and strong communication.

References

1. Berlot, Michele, and Janche Sang. "Dealing with Process Overload Attacks in UNIX." *Information Security Journal: A Global Perspective* 17, no. 1 (March 24, 2008): 1–1. https://doi.org/10.1080/19393550801929547.
2. "Code Hardening (Obfuscation & Encryption)." Guardsquare, December 6, 2019. https://www.guardsquare.com/en/mobile-application-protection/code-hardening-obfuscation-encryption.
3. Mourad, A., Debbabi, M., & Laverdière, M.-A. (2006). Security hardening of open source software. *International Conference on Privacy, Security and Trust: Bridge the Gap Between PST Technologies and Business Services* (pp. 2-4). Ontario: ACM.
4. Scarlett, Brittney. "8 Reasons Why Hardening Sprints Are Truly Necessary." Sagepath, July 6, 2017. https://www.sagepath.com/blogs/strategy/articles/8-reasons-why-hardening-sprints-are-truly-necessary.

5. Udupa, S.K., Debray, S.K., Madou, M., *Deobfuscation: Reverse Engineering Obfuscated Code*, IEEE 12th Working Conference on Reverse Engineering (WCRE), Pittsburgh, PA, Nov 7-11, 2005.
6. Xu, Hui, Yangfan Zhou, Yu Kang, and Michael R. Lyu. "On Secure and Usable Program Obfuscation: A Survey." *Computing Research Repository* abs/1710.01139 (October 3, 2017): 1–2. http://arxiv.org/abs/1710.01139.

CYNTHIA C. FRY
Professor Fry is currently a Senior Lecturer of Computer Science at Baylor University. She worked at NASA's Marshall Space Flight Center as a Senior Project Engineer, a Crew Training Manager, and the Science Operations Director for STS-46. She was an Engineering Duty Officer in the U.S. Navy (IRR), and worked with the Naval Maritime Intelligence Center as a Scientific/Technical Intelligence Analyst. She was the owner and chief systems engineer for Systems Engineering Services (SES), a computer systems design, development, and consultation firm. She joined the faculty of the School of Engineering and Computer Science at Baylor University in 1997, where she teaches a variety of engineering and computer science classes, she is the Faculty Advisor for the Women in Computer Science (WiCS), the Director of the Computer Science Fellows program, and is a KEEN Fellow. She has authored and co-authored over fifty peer-reviewed papers.

ZACHARY STEUDEL
Zac is currently a Junior Computer Science student at Baylor University. In Summer 2019, Zac worked as a Software Development Engineer Intern at Amazon and will be working as a Software Engineer Intern with Microsoft in Summer 2020. At Baylor, he is the teaching assistant for CSI 2334, "Introduction to Computer Systems," a low-level computing course taken by all Sophomore Computer Science students. Zac also greatly enjoys swimming and is currently the head coach for the Baylor Swim Club, conducting swim practices daily for the team and coordinating competitions with other Texas swim clubs.