Instructor: Matt Fredrikson TA: Tianyu Li

Due date: checkpoints on 4/22 and 4/29, final on 5/4; all deadlines at 11:59pm
Total points: 100

Lab 3: Break It then Fix It

1 Introduction

In Labs 0-2 you have taken steps to ensure that the data supporting your server remains confidential and safe from unwanted interference, all while adding more complex functionality to the server's implementation. This has been a constructive effort, focusing on principles of secure development rather than adversarial considerations.

In this lab, you will become the adversary for other students in the class. You will be assigned the implementations of Labs 0-2 for three other students, and your job is to find and exploit flaws in these implementations that lead to real security violations. You are encouraged to undertake this task using whatever means possible, short of communicating with the original author or modifying their implementation to insert a bug. You should use tools and techniques discussed in class, as well as finding ones that we did not discuss that are helpful in identifying vulnerabilities.

The vulnerabilities that you identify *must be exploitable*. You will not receive credit for bugs that impact the correctness of the server in carrying out its specified task, or its availability. For example, no credit will be given for finding a bug that causes the server to crash without showing how an attacker can further leverage it to either: *i)* take control of the server process; *ii)* overwrite memory containing the results of user-provided scripts, or memory containing authentication data; or *iii)* exfiltrate data from the server that is otherwise unavailable to an attacker. On the other hand, a memory safety bug that an attacker can use to overwrite the server's internal extendible hash data structure containing the current database state is a real vulnerability.

You should consider vulnerabilities in any part of the implementations you are assigned as fair game. This includes the assigned portions written by the student as well as the parts that were handed out with the lab, such as code in `tiny.c`. The exploits that you provide should violate either *confidentiality* or *integrity*. This applies both to data managed by the server, as well as the underlying platform that it runs on.

- Exploits targeting confidentiality should demonstrate that it is possible to use a flaw to exfiltrate data that is not made available by the intended functioning of the server. This could apply to data in the server's `.db` files, as long as the acquisition of that data violates the information flow policy of Lab 2 (and the vulnerability is in Lab 2). Likewise, this could apply to data in a file elsewhere on the system, as a correct functioning server should not allow one to read data from arbitrary locations in the filesystem.
- Exploits targeting integrity should demonstrate the possibility of changing server state in a manner that is inconsistent with the server's intended functionality. For example, an integrity exploit might show that it is possible to erase rows in a `.db` file without having called `undef` in a script. Integrity exploits can affect other parts of the system as well, such as by showing that one can leverage a bug to delete files or kill processes other than the one running the

server.

If you are uncertain that the exploit you have in mind addresses a real security issue, then please ask a **public** question on Piazza.

Learning goals. As you complete this lab, you will:

- Undertake a security analysis of other peoples' code, using independent research that builds on the background and skills you have developed in this class.
- Understand the distinction between bugs that matter for correctness and exploitable vulnerabilities by finding the latter in implementations of Lab 0 through Lab 2.
- Practice designing strategies for mitigating security issues.
- (Optional) Gain experience using popular tools to aid in finding vulnerabilities.
- Learn to write clear reports of your security analysis and findings so that others can reproduce your work.

Evaluation. This lab is worth 100 points, and grades will be assigned based on the reports that you turn in. You must find three vulnerabilities, successfully exploit them, and report on each one individually. If you cannot find three vulnerabilities, then for any implementation for which you were unable to find a vulnerability you must hand in a report that details the steps you used in your search, summarizes your findings, and convincingly explains why vulnerabilities are unlikely to be found.

Each report is worth 33 points (everyone gets 1 point by default for handing all three reports in). The breakdown given below applies to each individual report. The overarching principle that you should use to guide your writing is to **strive for reproducibility**. The course staff should be able to replicate your findings based on the report that you turn in.

Identify a vulnerability (5 points). Your report should begin by identifying a vulnerability in one of the implementations you are assigned. State which implementation the vulnerability exists in, and summarize the vulnerability. Then detail the steps that you took to find it, including any tools that you used and how you made use of them. If you had to write code or make changes to the target implementation to find the vulnerability, then explain your code and/or the changes you made, and provide the code seperately.

Explain the security issue (8 points). After presenting the vulnerability and recounting your steps to find it, concisely explain why this vulnerability is relevant to security and not just a correctness or availability bug. Clear and convincing explanations will receive 8 points, but if the course staff concludes that the "vulnerability" is really just a correctness issue that is not exploitable, then you will not receive points for the other tasks either.

Explain how to exploit it (10 points). Provide a proof-of-concept exploit that successfully exploits the vulnerability to violate the security of the target implementation. This section of the report should illustrate the security issue from the previous section in concrete terms, and give the course staff enough information to exploit the vulnerability. Your explanation

should give detailed, step-by-step instructions that lead to a security violation, and provide any necessary code.

Propose a fix (10 points). The final section of your report should propose a fix for the vulnerability you found and exploited. Your fix should eliminate the possibility of successful exploitation, without interfering with the correct functioning of the implementation or introducing additional bugs or vulnerabilities. On the other hand, if aspects of the implementation were already buggy in ways that affect correctness, then your fix does not need to address those pre-existing bugs. Your fix may or may not use techniques that have been discussed in this course, and you are free to propose any approach as long as you can justify its efficacy. You do not need to implement the fix yourself, but your proposal should contain sufficient detail to allow the original author to implement it without doing unnecessary work.

(Alternate) No vulnerability (33-99 points). If you are unable to find three vulnerabilities, then you must provide a detailed account of your efforts on each set of implementations for which search was unsuccessful. For example, suppose that you were assigned the implementations of Students X, Y, and Z, and were only able to find a vulnerability in one of the implementations of Student X. Then you should hand in two additional reports detailing the steps that you took in searching for vulnerabilities in Labs 0-2 of Students Y and Z. These reports will be graded as a whole, for a total possible number of 66 points. Your reports should convince the course staff that you exercised due diligence in your search, leading to the conclusion that there are no readily-exploitable bugs in the implementations you were given.

(Extra credit) Implement the fix (5-10 points). If another student finds a vulnerability in your implementation, you have the opportunity to fix it for a variable amount of extra credit. The amount of extra credit that you receive depends on the complexity of the fix, as well as the efficacy of your implementation of it.

What to hand in. For each report that you write, save it in a PDF file (note: do *not* hand in Word documents or text files!) titled `report-N.pdf` (where N is 1,2, or 3), and put it in a folder named `exploit-N` along with any code or additional documentation necessary for reproducibility. Archive the three folders in `<andrewid>-lab3.zip`, and hand in on Gradescope.

The final deadline for this lab is **Friday, May 4**, and there will be **no exceptions** as this is the last day of classes for the semester. Additionally, you must hand in one of the reports by **Sunday, April 22 at 11:59pm**, and the next by **Sunday, April 29 (11:59pm)**. This is to allow the author of the corresponding implementation the opportunity to implement a fix for extra credit.

2 Helpful pointers and tools

To get started on this lab, it is advisable to first look over the implementations you are assigned to familiarize yourself with the layout and approach that its author took. Because you already implemented the same functionality yourself, you probbably have some idea of where the interesting bugs might live, and how you might exploit them. Before attempting to use any tools or a sophisticated technique, try simple things manually first.

We have covered a number of tools and techniques throughout the semester that can help you accomplish the tasks in this lab. Tools like CBMC and Pin can be directly applied to these and related tasks, as is documented in the following articles. Note that this is not a complete list, and you are encouraged to look through the related work sections as well as search other Internet sources for more.

- Adi Sosnovich, Orna Grumberg, Gabi Nakibly. *Finding Security Vulnerabilities in a Network Protocol Using Parameterized Systems.*
- Jonathan Gallagher, Robin Gonzalez, Michael E. Locasto. *Verifying security patches.*
- Rafal Wojtczuk. *UQBTng: a tool capable of automatically finding integer overflows in Win32 binaries*.
- Pasquale Malacaria, Michael Tautchning, Dino DiStefano. *Information Leakage Analysis of Complex C Code and Its application to OpenSSL.*
- Dawson Engler, Madanlal Musuvathi. *Static Analysis versus Software Model Checking for Bug Finding.*
- Gal Diskin. *Binary Instrumentation for Security Professionals.*
- Jonathan Salwan. *Taint analysis and pattern matching with Pin.*
- Jonathan Salwan. *In-Memory fuzzing with Pin.*
- Jonathan Salwan. *Concolic execution with Pin and Z3.*
- Jonathan Salwan. *Stack and heap overflow detection at runtime via behavior analysis with Pin.*

These references are meant to get you started researching vulnerability analysis techniques, and only apply to the tools we have used in class. You may find some of them more directly pertinent to the task at hand than others, and your mileage will almost certainly vary, so you should avoid spending too much time reading all of them thoroughly. As you find additional references, please remember to share those that were especially helpful with the rest of the class on Piazza.

You are also encouraged to explore tools outside of those that we covered in class. Below is an incomplete list of potentially useful tools to consider. Before attempting to use any of them, it is a good idea to do a bit of background reading to understand what the tool is capable of, how much effort is required to effectively use it, and what sorts of vulnerabilities it might uncover. It is best to have a game plan before using a tool, with a reasonable goal in mind of what you hope to accomplish. Finally, if you have already invested some time in a tool and have yet to see returns, consider moving on and trying something different rather than potentially wasting more time.

- Klee is a symbolic execution engine for C and C++. This is a versatile tool that can be applied to finding vulnerabilities as well as other bugs. You might find it enlightening to read EXE: Automatically Generating Inputs of Death before deciding whether to use this tool.
- American fuzzy lop is a fully-automated security-oriented fuzzer. The main tool `afl-fuzz` will not tell you whether a bug is exploitable, just whether it crashes the program. Consider installing the useful afl-utils suite of tools for additional functionality that may help you discover vulnerabilities more quickly.
- Facebook Infer is a static analysis tool that is particularly good at finding memory issues. Note that while it is not designed specifically for exploitable bugs, you may still find it useful.
- Flawfinder is a simple tool for finding potential security bugs in C/C++ programs.
- CPAchecker is a static analysis platform for C and C++.
- Triton Dynamic Binary Analysis framework. Built on top of Pin, provides a symbolic execution engine, taint analysis, and other useful APIs. Has Python bindings for rapid prototyping.
- Jonathan Salwan's Pin tool collection has several useful tools for vulnerability-related tasks. You might consider adapting one of these to suit your needs.
- RATS is an automated scanner for finding potential security bugs in a number of languages.
- Peach is a fuzzing framework maintained by Mozilla that is more configurable than American fuzzy lop. Compatible with Python.
- Splint is a lightweight static analysis tool for C programs. It focuses particularly on security bugs, and is fairly easy to use.

As with helpful references, if you encounter a tool that is not listed here and obtained good results from it, please share with the rest of the class on Piazza.