EN.650.660, Software Vulnerability Analysis, Fall 2021 Instructor: Reuben Johnston, reub@jhu.edu

Homework 4

See Blackboard for grade %

Due Date

See Blackboard for date

Do not wait until the last minute for starting this assignment. Reverse engineering and exploitation activities, even simple ones such as these, can be very tedious and time consuming.

Summary

Figure 1-Angband

This exercise is practice for binary reverse engineering using Ghidra and format string vulnerability exploitation. Techniques used in the class exercises will assist in performing this homework assignment. It is strongly recommended to first conceptually think through all the exploit sequence steps prior to working on the code for generating the tainted variables.

It is encouraged to collaborate with your students in discussions on the concepts involved with this exercise. However, do not store any code online or share code with other students.

When you are planning, be sure to have a look at the specified grading rubric to get a feel for what is required.

Requirements

Executive summary briefing or report

These summaries (2-3 page report or 5-10 minute briefing) are intended to be reference material for when you are performing threat modeling in practice and must contain an overview of the project, the project goals, project execution highlights (i.e., what was performed, with pertinent technical highlights), what goals were accomplished (and any that were not achieved), and what was learned from the project. For the report format, students shall use the format required by the IEEE Open Journal of the Computer Society (template is available on Blackboard, see Instructions page, and was downloaded from here). For the presentations, you may choose your own style (PowerPoint is recommended) but you must submit recordings via Panopto and provide digital copies of the slides. Summary reports and slides for

EN.650.660, Software Vulnerability Analysis, Fall 2021 Instructor: Reuben Johnston, reub@jhu.edu
summary briefing are submitted on Blackboard with the deliverables. Summary briefing recordings are submitted on Blackboard via Panopto.

Reverse engineering and exploitation deliverables

For this assignment, place exploit source code and compiled binaries in a single zip file submitted for the deliverables. Provide answers to any questions and requests outlined below in a Microsoft word document that is also included in the deliverables zip file.

- Prepare your VM by installing dependencies and setting up environment
  - $ sudo apt-get install libncurses5-dev libncursesw5-dev libx11-dev libsdl1.2-dev libsdl-ttf2.0-dev libsdl-mixer1.2-dev libsdl-image1.2-dev
  - Extract the provided angband.tar into /usr/games (e.g., $ cd /usr/games && sudo tar -xvf ~/Downloads/angband.tar)
  - $ sudo chown $USER:$USER /usr/games/angband
  - Copy the provided binary into /usr/games/angband/games
- Test run the binary
  - E.g., $ /usr/games/angband/games/angbandv1_stripped -g -msdl -ssdl
- Import the 64-bit linux binary into Ghidra
- What are the imported functions used by the binary? Provide their names below.

libc, libm, libncursesw, libSDL, libinfo, libX11

Deliverable 1

- Identify the backdoor function (the one where you pass in the tainted variables) and secret key to unlock it using commands in the program. Provide the hexadecimal offset in the binary for the function and the details for unlocking below.

secret key: % offset: 0020f182

Deliverable 2

- Name the backdoor function in Ghidra and save the database. Provide a screenshot of the renamed function in Ghidra below.

screenshot provided - Dev 2.png

Deliverable 3

- Identify the vulnerable function (the one with the achievable overflow). Provide the hexadecimal offset in the binary for the function below.

0020f2D6

Deliverable 4

- What was the design mistake for this vulnerability? Provide your answer below.

the user is able to input format specifier to printf

Deliverable 5

- Name the vulnerable function in Ghidra and save the database. Provide a screenshot of the renamed function in Ghidra below.

screenshot provided - Dev 6.png

Deliverable 6

- Reverse engineer the local variables for the vulnerable function and provide details below (list a suitable name and/or description for each). Provide a screenshot as well below of the renamed local variables in Ghidra.

screenshot provided - Dev 6.png

Deliverable 7

- Debug the program using gdbserver in Eclipse by using a C/C++ remote target launch configuration
  - Note: there are a couple of useful gdb commands at the end of this document
- Set a break point in the vulnerable function and get the function to run. Inspect the stack frame. Provide a screen shot of the stack frame below.

screen shot provided - Dev 8.png

Deliverable 8

- Implement the Dumb Leak attack that reaches the vulnerable code by constructing nine tainted variables using a wrapper.c file that generates them
  - Format for the nine tainted variables (8 dummy addresses and buffer for format string) is the following
    - Dummy addresses 1-8 input are 8-digit hexadecimal numbers
    - taintedbuffer input is a char array for a format string
  - Run the vulnerable program in the debugger and get to the backdoor mode
  - Then, copy and paste your crafted variables into the debugger console's stdin and continue the program by stepping over the vulnerable line and watch the variables change on the stack
    - Take a screenshot of leaking stack data from your program

Screenshot provided - Dev 9.png

Deliverable 9

- Similar to Deliverable 9, construct nine tainted variables (using a wrapper.c file that generates them) to implement the arbitrary read attack that reaches the vulnerable code (note, this attack provides one valid address, 7 dummy addresses, and the buffer).
  - Modify the code from Deliverable 9 and disable the dumb leak section using #if 0 and #endif preprocessor directives
  - Code the arbitrary read portions in a new section using #if 1 and #endif preprocessor directives
    - Take a screenshot of reading an arbitrary string at a memory address in the VM space (e.g., environment variable section has many strings in it) for the program with your exploit and provide it below.

Screenshot provided - Dev 10.png

Deliverable 10

- Similar to Deliverable 9, construct nine tainted variables (using a wrapper.c file that generates them) to implement the sledge-hammer attack that reaches the vulnerable code (note, this attack provides 8 dummy addresses and buffer for format string).
  - Modify the code from Deliverable 9 and disable the arbitrary read section using #if 0 and #endif preprocessor directives
  - Code the sledge-hammer portions in a new section using #if 1 and #endif preprocessor directives

- Take a screenshot of crashing the program with your sledge hammer and provide it below (For entertainment purposes, I recommend running this video in the background and imagine that your exploit is singing this song to the vulnerability).

Deliverable 11

- Similar to Deliverable 9, construct nine tainted variables (using a wrapper.c file that generates them) to implement the arbitrary write attack that reaches the vulnerable code (note, this attack provides eight valid addresses and the buffer) and launches shell code.
  - Modify the code from Deliverable 9 and disable the sledge-hammer section using #if 0 and #endif preprocessor directives
  - Code the arbitrary write portions in a new section using #if 1 and #endif preprocessor directives
    - Take a screenshot of executing your shell code after your exploit corrects the stack frame and provide it below.

Screenshot provided - DEN 12.png

Deliverable 12

9) What was the design mistake for this vulnerability?

the user is able to input format specifier to print-f

Deliverable 13

10) For each Deliverable 8-11, what would be the specific effects to your exploit if the security measures were reenabled? E.g., what if the program were built or run with ASLR enabled, stack protection (canaries) enabled, non-executable data sections in memory, and glibc format string security checks and Fortify source protections (see notes for details on Fortify source protections)?

Effects on Deliverable 8:

Effects on Deliverable 9: Refer DEN 10.txt

Effects on Deliverable 10:

Effects on Deliverable 11:

Deliverable 14

Grading Rubric

Executive summary (50% overall)

Table 0.1-Executive report grading rubric (2006, Leydens, Santi)

Objective	1 - Exemplary	2 - Proficient	3 - Apprentice
Format / layout / organization	Report is very clear, coherent with excellent transitions	Report is clear and coherent, strong throughout	Report has some gaps, some weak sections

Writing mechanics	Report is virtually error-free, and contains few if any reader distractions	Report is logical and easy to read, and may contain a few errors causing minimal reader distraction	Report is generally clear, but distracting errors and flow make it difficult to follow at times
Figures / Tables	All figures and tables are easy to understand, and are clearly linked to the text. Story can be told almost entirely through figures.	All figures and tables can be understood with information given and are linked to text. One or more need improvement. May need more figures to tell the story.	Figures and/or tables are hard to understand, are not all linked to text. Several need improvement. Several more figures are needed to tell story.
References	All sources identified and referenced appropriately. Evidence of careful and thorough research for outside information.	All sources identified and referenced appropriately. Includes mostly readily available works.	All sources identified. Only readily-available works included. Some weaknesses in referencing, such as missing publisher information.
Typical Grade (average):	90-100%	80-90%	70-80%

Table 0.2-Executive briefing grading rubric

Objective	**1 - Exemplary**	**2 - Proficient**	**3 - Apprentice**
Format / layout / organization	Brief is very clear, coherent with excellent transitions	Brief is clear and coherent, strong throughout	Brief has some gaps, some weak sections
Slides	All figures and tables are easy to understand, and are clearly linked to the brief. Story can be told almost entirely through figures.	All figures and tables can be understood with information given and are linked to brief. One or more need improvement. May need more figures to tell the story.	Figures and/or tables are hard to understand, are not all linked to brief. Several need improvement. Several more figures are needed to tell story.
References	All sources identified and referenced appropriately. Evidence of careful and thorough research for outside information.	All sources identified and referenced appropriately. Includes mostly readily available works.	All sources identified. Only readily-available works included. Some weaknesses in referencing, such as missing publisher information.

Typical Grade (average):	90-100%	80-90%	70-80%

Programming deliverables (50% overall)

- To get 100% you will need to complete all the above items.
- Getting the stack-leaking demonstration (Deliverable 9) working is worth 80%
- Getting the arbitrary read demonstration (Deliverable 10) working is worth 5%
- Getting the sledge-hammer demonstration (Deliverable 11) working is worth 5%
- Getting the arbitrary write demonstration (Deliverable 12) working is worth 5%
- Answering the other questions is worth 5%

References and useful resources

- See the reverse engineering (bomb lab) class exercise on BlackBoard.
- See the debugging exploits class exercise on BlackBoard.
- See the format-string exercise on BlackBoard

Useful GDB commands

- To locate the .text base address using gdb:
  (gdb) maintenance info sections
- Set a breakpoint on an address using gdb:
  (gdb) b *0xdeadbeef

Fortify source

- Fortify source is a gcc optional feature that is enabled using: -D_FORTIFY_SOURCE
- There are some notes from the original patch to gcc here