Cyber Grand Challenge Services

1 Overview

A selection of over two hundred vulnerable network services created by DARPA for their Cyber Grand Challenge (CGC)[1], are available for your exploration and experimentation. Each of these services include one or more deliberately planted vulnerabilities. The vulnerable services were adapted from the CGC corpus to run within Linux by Trail of Bits[2].

While these services were created by DARPA for an automated capture-the-flag competition, they represent a good collection of memory corruption vulnerabilities that students may explore to better understand the properties and mechanics of different types of software flaws. This lab runs the services in a typical client-server environment over a network as *inetd* services.

This collection of services is intended to provide you with examples to explore using static program analysis tools as well as debuggers and observation of network traffic.

1.1 Background

The student is expected to have an understanding of the Linux command line, and some amount of low level programming. It is expected that the student will have some experience with debuggers and decompilers.

2 Lab Environment

This lab runs in the Labtainer framework, available at http://nps.edu/web/c3o/labtainers. That site includes links to a pre-built virtual machine that has Labtainers installed, however Labtainers can be run on any Linux host that supports Docker containers.

From your labtainer-student directory start the lab using:

```
labtainer cgc
```

A link to this lab manual will be displayed.

The lab includes two computers, a client and a server.

2.1 Server

The *server* runs all of the services as *xinetd* services. This means the services do not execute until a TCP connection is made to the port associated with the service. Each TCP connection gets its own instance of the service. The *client* computer has a network connection to the server, and it includes utilities for interacting with services.

The executable binaries for the services are in `/usr/sbin`, as can be seen in the service files found in `/etc/xinetd.d/`. The server includes the Ghidra tool and IDA Free.

Source code for services is in the `$HOME/challenges` directory. Each service includes a `README.md` file that describes the service and its vulnerabilities.

[1]`https://www.darpa.mil/program/cyber-grand-challenge`
[2]`https://github.com/trailofbits/cb-multios`

2.2 Client

The client computer includes utilities and data sets for interacting with the services. These utilities refer to a `service.map` file that map service names to their TCP port numbers. If you create your own scripts or programs to interact with services, consider using that map.

The client includes wireshark for viewing network traffic.

2.2.1 Service polls

The `poll.py` script will interact with the service to confirm it is operational (and provide you with sample service traffic to help you understand the functions of the service). The client includes five polls per service, and these are named as the integers 1-5. For example

```
./poll.py payroll 2
```

will use the second poll to interact with the payroll service.

2.2.2 Proofs of vulnerabilities

The client includes one or more *proof of vulnerabilities* (PoVs) that exploit software flaws identified in the service `README.md` file. These *PoVs* have one of two forms:

- Type I – Causes the service to crash with a SEGV. These crashes can be observed in the `/var/log/messages` file, or using gdb.
- Type II – Causes the service to leak 32 bits of "secret" data from a page of memory filled with random bytes. The PoV reports this data when successful.

The `pov.py` utility on the client executes the PoVs. For example, the following will throw the first POV against the payroll services:

```
./pov.py payroll 1
```

Use of the `-d` option will delay network reading and writing for the given number of seconds. This is intended to let you attach to the service using gdb.

3 Tasks

This exercise is intended to be self directed and at your own pace. The suggestions below may aid your exploration.

3.1 Review services

Browse through the services by reading the `README.md` files in the `$HOME/challenges` directory on the server. Perhaps start with "easy" services such as `Palindrome`. Many of these services were part of the the CGC Final Event. You can learn about which of those services were successfully exploited during the CGC from this web page: https://www.lungetech.com/cgc-corpus/cfe The service identifiers on that page can be (mostly) mapped to their "common names" using the `$HOME/common_names.txt` file on the client.

The service source code can be viewed along with the `README.md` files. Note the source code uses `ifdef` constructs to either introduce or patch each vulnerability. The patched instances of each executable have the same name as the vulnerable instance within the `/usr/sbin` directory, but with a `_patched` suffix.

Comparing the vulnerable source code to the patched source code, and comparing vulnerable binaries to patch binaries, can be very helpful toward understanding different types of software flaws.

At this point it is worth repeating that this lab is about exploration. No attempt is made to hide information from the student. It is entirely up to you to hide information from yourself. For example, pick a service and try to understand its behavior without referring to the source code.

3.2 Disassemble using Ghidra

If you are not familiar with Ghidra, consider performing the Labtainers `ghidra` lab. The Ghidra tool is installed on the server, and can be started using the `./ghidra` command in the home directory. Your default project directory is `$HOME/mystuff/ghidra`, which allows your Ghidra data to persist beyond the life of the server container. Create a project with the name of one of your selected services. Once you've created the project, use the `File/Importfile` menu option to import the executable binary from `/usr/sbin`. Then double-click on the newly imported file and direct Ghidra to perform analyis.

You will note that the resulting analysis includes a symbols (e.g., helpful variable names). Consider challenging yourself by creating and analyzing a *stripped* version of the binary using the Unix `strip` command.

3.3 Disassemble using IDA Free

If you have not yet install IDA Free in a Labtainers lab, you can do so by running the `idafree70_linux.run` program in your home directory on the server. Your home directory will have a idafree-7.0 subdirectory which is shared with the host, thereby letting you share this IDA installation with other Labtainer labs. After installing IDA, use the `./ida` command in your home directory to start IDA. For example:

```
./ida /usr/sbin/payroll
```

NOTE: If windows appear black or as noise, try resizing them. If they do not resize, closing them usually works.

When you first open an executable, IDA will display an error message telling you to choose a different directory for the database. Select the `$HOME/mystuff/ida` directory so that your IDA databases persist. After you've used IDA to analyze a given executable, you can open it later by giving IDA the name of the database, e.g,

```
./ida mysuff/ida/payroll.64
```

3.4 Observe network traffic

Start wireshark on the client and capture traffic on eth0. Then run a poll, e.g.,

```
./poll.py payroll 1
```

Note that some services use a binary protocol, making it more challenging to understand by looking at wireshark captures.

Poll traffic is generated by XML files found in `$HOME/challenges/<service>/polls`. View those files to help you understand the protocols. Consider modifying these files to alter the interaction with the service. You may also add a `delay` directive to delay traffic while you attach a debugger to the service, e.g., add:

```
<delay>10000</delay>
```

as the first item in the `replay` section of the XML to delay for ten seconds, during which time you could issue this command in the server:

```
gdb -p `pgrep payroll`
```

Note that gdb is configured (in the `/etc/gdb/gdbinit` file), to provide you with source code. Consider trying debug sessions by first moving the source code (or changing the directive) so that you cannot see the source from gdb. Also consider using the `strip` command to remove symbols from some executables.

3.5 Review PoVs

The PoVs available on the client are artifacts from the CGC, in which the authors of vulnerable services were required to create example PoVs for each of the deliberately planted service vulnerabilities[3] . These are executable x86 binaries whose source code can be found along side the service source code on the server. The lab environment does not support recompilation of the PoVs. Consider creating your own tool for generating PoVs. You can then use the PoV source code to reproduce and expand on the sample PoVs, e.g., to take control of an exploited service.

3.6 Debug service during exploit

Use the `pov.py` command to exploit a service. For example,

```
./pov.py payroll 1 -d 10
```

And then run this on the server to attach gdb to the service:

```
gdb -p `pgrep payroll`
```

If you direct gdb to continue the process, you will see a SEGV (for Type I PoVs). You can also observe the crash in the `/var/log/messages` log. Type II PoVs do not crash. They leak "secret" information from a specific memory page and continue. The output of the `pov.py` function will tell you the location of the page, and the value that was leaked. It does not tell you the address of the leaked value, for that you might use scripting within gdb to find the address of the leaked value, and then set a hardware breakpoint on reads of that address to find where the leak occurs.

Consider configuring gdb to not include source code, and the use of stripped binaries to give you a better feel for real world analysis.

3.7 Reverse a patch

When vendors release patched versions of applications, those patches can be compared to unpatched versions of the software to understand the flaw being patched. Without referring to source code, and using stripped binaries, compare the vulnerable executables to the patched instances. Attempt to use that comparison to create your own PoV.

[3]Services also include unintended vulnerabilities common to most software development

3.8 Limitations and notes

The CGC included some vulnerable services that ran as two or more processes, communicating through shared pipes. This lab does not include any of those services.

The CGC Archive site includes PoVs and patched services submitted by the automated competitors during the competition. Those executables are linked to run within the DECREE execution environment, and thus do not run on Linux.

4 Submission

After finishing the lab, go to the terminal on your Linux system that was used to start the lab and type:

```
stoplab
```

When you stop the lab, the system will display a path to the zipped lab results on your Linux system. Provide that file to your instructor, e.g., via the Sakai site.