Homework 1

EPFL COM402 — Information Security and Privacy 2019

- In this homework, you will be collecting secret tokens, one for each exercise. Please save these tokens, as you need to submit them on Moodle to get points. The tokens are unique for each student. See the format of the submission on Moodle. For your convenience, we provide an online script to check whether your tokens are correct.
- You need a Docker environment to be set up on your machine.
- There are 5 exercises in this assignment, with the following grading: **ex 1 (10 points), ex 2 (10 points), ex 3 (20 points), ex 4 (20 points), ex 5 (40 points). The token for an exercise gives you full exercise points, while not having the token gives you 0 points for that exercise.**
- The homework is due on **Sunday, March 17, at 23h55 on Moodle**.
- You can resubmit your tokens as often as you want until the deadline. **Late submissions, however, are not allowed**. Thus, we advise you to submit early and resubmit in case you obtain more tokens, so that you do not miss the deadline by a few minutes.

(10p) Exercise 1: Are Clients Better Authenticated In The West Or In The East?

To discover your first token, you need to pass the authentication on the DeDiS Secure Login page http://com402.epfl.ch/hw1/ex1

You must use **your personal EPFL email address** as a login name. As soon as you are logged in, you will be able to see your token! The problem is that you have no clue what your password is. But maybe you can find out how the password-verification is performed and trick it? **Please do NOT insert your GASPAR password in this exercise!**

(10p) Exercise 2: Is After-Lunch Time Best for Hacking?

The Evil Corp is on track again. You do not like many things about them but the worst one is that, if you log in once, they start tracking you. Even if you leave and come back as a new person, they will recognise you. Fortunately, you have heard rumours that their authentication mechanism is severely broken, and it is not so difficult to log in as a simple user. But can you go even further and spy on the whole world?

The login web page of Evil Corp is located at http://com402.epfl.ch/hw1/ex2/login. If you manage to spy on the whole world, you will see your secret token on the web page.

Use again your personal EPFL email address as a username. And, as in the previous exercise, **using your GASPAR password here is NOT the best idea**.

(20p) Exercise 3: All their parcels are belong to you[1]

In this and the following exercise, you need to imagine that you are in a cyber-cafe. The kind of cyber-cafe with a scent of of freshly brewed coffee but also crowded with computer newbies who come here to browse the Internet and shopping websites.

You have managed to take the control over the router in the cafe and, as a result, all the traffic in the cafe goes through your machine so you can intercept and modify HTTP packets. Not bad!

You notice that your neighbour in the cafe has added some product in a shopping basket on one of the shopping websites. You find this product very cool and decide to have it delivered to you instead. Your goal is thus to **substitute the shipping address of the HTTP purchase request with your personal EPFL email address**. The other fields need to stay the same. The steps you need to follows are described below.

a. Setup

First, you need to create a new directory for this exercise (`mkdir`) and go (`cd`) to that directory. Then create an empty python script here, for example with:

```
touch interceptor.py
```

Then download the shell script `run_dockers.sh` from Moodle *to the same directory that you just created*. The script pulls and runs two Docker images:

- *`generator`* representing a machine of a random client in the cafe (in principle you shouldn't need to connect to this container);
- *`attacker`* representing your machine, on which you'll be running the code you write.

The script creates a mapping from your current directory to `/shared`. You can then run the script as follows: `sh run_dockers.sh` *`your_personal_epfl_email_address`*

The *generator* container periodically sends HTTP packets to two URLs:

- `com402.epfl.ch/hw1/ex3/shipping`
- `com402.epfl.ch/hw1/ex4/transaction`

To make it simpler for you, the *generator* container sends all the necessary information in a couple of minutes and then just repeats it again in a random order. The exact time a packet is sent is randomly chosen within that interval.

You can now connect to the *attacker* docker container:

[1] Inspired by https://en.wikipedia.org/wiki/All_your_base_are_belong_to_us

```
docker exec -it attacker /bin/bash
```

Since the `attacker` container is in fact the router, it must do proper network address translation (NAT). To enable NAT you can use `iptables`. `Iptables` is an user-space application program that allows a system administrator to configure the tables provided by the Linux kernel firewall. To enable NAT in the `attacker` container, type the following:

```
docker exec attacker /bin/bash -c 'iptables -t nat -A POSTROUTING -j MASQUERADE'
```

Since the attacker's goal is to intercept traffic, we now need a way to tell the Linux kernel on the attacker machine to give us, i.e., to user space, all packets it intercepted from the generator container. To do that, we first set up an `iptables` rule that redirect some specific packets to a special user-space queue called NFQUEUE. To enable it:

```
docker exec attacker /bin/bash -c 'iptables -A FORWARD -s 172.16.0.2 -p tcp --dport 80 -j NFQUEUE --queue-num 0'
```

- Every packet that goes through the attacker machine coming from the generator matches the iptables FORWARD chain. In our case the generator sends packets to `com402.epfl.ch`, but the packet passes through the attacker, since the attacker has control of the router.
- For theses packets, check if the source address is the one from the generator, and if it's a TCP packet with the destination port being 80 (i.e., mostly HTTP traffic).
- In that case, redirect that packet to a user-space queue: NFQUEUE number 0.

All packets in the NFQUEUE can be read by user space programs which can decide if a packet goes on or is dropped.

b. Seeing traffic

This section explains you how can you programmatically read the traffic. Please note that **all libraries needed are already installed inside the attacker container**.

Netfilter

In order to read from this NFQUEUE in Python3, we need to use the `NetFilterQueue` library. All documentation can be found at https://pypi.python.org/pypi/NetfilterQueue .
Some notes on `NetfilterQueue`:

- Follow the first example, DON'T bind over a socket.
- You need to bind to the queue number 0
- To not overload the queue, pass a third parameter to the bind method specifying what should the length of the queue, as follows: `nfqueue.bind(0, callback, 100)`
- To get the raw bytes, call `pkt.get_payload()`
- To drop a packet: `pkt.drop()`. To let a packet go through: `pkt.accept()`.

You can test your script by running `python3 shared/interceptor.py` from **inside** the `attacker` container or by running from your machine:

```
docker exec -it attacker python3 shared/interceptor.py
```

Scapy

Now that you can see all packets within the callback function, you need to parse them. Indeed, `netfilter` will give you RAW packets as bytes, containing all layers from the Ethernet frame to the Application layer (see https://en.wikipedia.org/wiki/Data_link_layer for more information).

`Scapy` is a very powerful packet manipulation Python library. It's easy to parse packet and modify traffic with it. The official documentation is at http://www.secdev.org/projects/scapy/doc/ . A more "practical" tutorial can be found here: https://bt3gl.github.io/black-hat-python-infinite-possibilities-with-the-scapy-module.html .

Since the goal isn't to become a Scapy expert, here's the list of methods you will need for this exercise:

- Create an IP packet object out of the raw bytes: `ip = IP(raw)`
- Check if there is a specific layer in the packet: `ip.haslayer(layer)`
- Get a specific layer: `ip[layer]`
    - Some useful layers: `TCP`, `Raw (application layer)`
- It's good to check if the destination port is 80 on the TCP layer:
    - `tcp.dport == 80`
    - Question: Why would there not be any application layer in a TCP packet ?
- Get the application layer payload: `http = ip[Raw].load.decode()`

HTTP

Now that you have the http payload, you need to parse it! As a reminder, here's a generic HTTP traffic payload containing JSON:

```
POST /hw1/ex3/shipping HTTP/1.1
Host: com402.epfl.ch
Content-Length: 91
Content-Type: application/json
User-Agent: Dumb Generator

<JSON CONTENT>
```

In python, `json.loads(data)` will prove useful!

Troubleshooting

If you don't see the victim packets coming, or the intercepted messages don't have any content:

- Make sure you followed the instructions correctly: bind to queue 0, do not bind to a socket, accept or drop every packet
- Try sudo: `sudo sh run_dockers.sh <...>`, `sudo docker exec <…>`
- Try *permanently* disabling SELinux on your machine, if it is enabled.

c. Sending traffic

Once you successfully parsed and changed the shipping address in the HTTP payload, you can send **a new packet** containing the right json directly to the server, specifically to to `com402.epfl.ch/hw1/ex3/shipping`. In this exercise, you can use the requests library (documentation at http://docs.python-requests.org/en/master/).

Please note: There will be **multiple packets** with a shipping address in their HTTP payload, **but they are all identical** (so that missing one shipping packet does not require to restart the generator). **Thus if you have intercepted one such packet, you do not need to search for more.**

If you do everything correctly, the shopping website at `com402.epfl.ch/hw1/ex3/shipping` will answer with an HTTP packet carrying `200` status code and a secret token in its payload.

(20p) Exercise 4: Keep Your Feet Warm And Sensitive Data Protected

You are still in the cyber-cafe, but now you realize that you can do much more than sending a product to your email address. People in the cafe are using their credit cards for payments over the Internet and passwords for logging in into their online accounts. And all this data is not encrypted! How convenient! You have to find a way to look at all the traffic and find some valuable information.

You'll use the same *generator* and *attacker* containers as in the previous exercise, but this time you need to search through the traffic you sniff. Your task is to find in the traffic **five distinct pieces of sensitive information, either credit card numbers or passwords**. Specifically, **some packets** contain in their HTTP payload sensitive information that has the following format:

`cc --- card_number`, where *`card_number`* is either `xxxx.xxxx.xxxx.xxxx` or `xxxx/xxxx/xxxx/xxxx` and `x` is a digit `0-9`.

or

`pwd --- password`, where *`password`* is a mix of digits `0-9`, upper-case letters `A-Z`, symbols `: ; < = > ? @` The passwords also have a length between 8 and 30 symbols. Correct passwords **DO NOT** contain lower-case letters.

Sensitive information can be preceded or succeeded by random strings containing a mix of digits `0-9`, upper-case letters `A-Z`, and lower-case letters `a-z`. The secret is always separated through a space from these random strings, one space before the sensitive word and one after. For example, an HTTP payload containing sensitive information is:

```
lKJdeL324bhvw34FbnFa        cc        ---        1234.5678.9012.3456
khj35jvDKldbFGhj6fre
```

But be careful! Some packets contain strings that resemble the format of secrets, but do not match the exact format description. Thus, they do not contain valid secrets and you should not extract information from them.

As soon as you collect all five distinct secrets, you need to create a single HTTP packet with a JSON payload of the following form:

```
{
“student_email”: your_EPFL_email_address ,
“secrets”: [secret1, secret2, secret3, secret4, secret5]
}
```

where you give your EPFL e-mail address as in the previous exercises and `secrets 1-5` with the secrets you have sniffed. Afterwards, you need to send the HTTP packet you have created to `com402.epfl.ch/hw1/ex4/sensitive`.

If you have found **all five secrets**, the server will respond with an HTTP packet carrying the `200` success code and your final secret token!

(40p) Exercise 5: Stack Smashing

The goal of this exercise is to gain hands-on experience with the effects of buffer overflows and other memory-safety bugs. Your goal is to understand the vulnerabilities in four target programs, and write an exploit for each target program.

Environment Setup

Since we didn’t want to deprive you of the pleasure of using a virtual machine (VM), for this exercise, you will test your exploit programs in a VM. Find the VM images for different platforms along with more useful information on how to use the VM here. Be sure to follow the instructions, and don’t forget to specify the port when calling `ssh` or `scp`.
Note that the link also contains the source code for the targets and skeleton source for the exploits, which you will be implementing. Speaking of which...

Targets

The `targets/` directory contains the source code for the targets, along with a Makefile for building them. Your exploits should assume that the compiled target programs are installed setuid-root in `/tmp` -- `/tmp/target1`, `/tmp/target2`, etc.
ma

To build the targets, change to the `targets/` directory and type `make` on the command line; the Makefile will take care of building the targets.

To install the target binaries in `/tmp`, run:

```
make install
```

To make the target binaries setuid-root, run:

```
make install
su
make setuid
```

Once you've run `make setuid` use `exit` to return to your user shell. Alternatively, you can keep a separate terminal or virtual console open with a root login, and run `make setuid` (in the `~user/hw1/targets` directory!) in that terminal or console.

Keep in mind that it'll be easier to debug the exploits if the targets aren't setuid. (See below for more on debugging.) If an exploit succeeds in getting a user shell on a non-setuid target in `/tmp`, it should succeed in getting a root shell on that target when it is setuid. (**But be sure to test that way, too, before submitting your solutions!**)

Exploits

The `sploits/` contains skeleton source for the exploits which you are to write, along with a Makefile for building them. Also included is `shellcode.h`, which gives Aleph One's shellcode.

You must use this shellcode, as this will be used in the grading scripts!

Assignment

Your goal in this exercise is to attack the targets. To do so, you are going to write exploits, one per target. Each exploit, when run in the VM with its target installed setuid-root in `/tmp,` should yield a root shell (`/bin/sh`). You can use the command `whoami` to verify whether you succeeded or not.

Hints

Read the Phrack articles suggested below. Read Aleph One's paper carefully, in particular.

To understand what's going on, it is helpful to run code through `gdb`. See the GDB tips section below.

Make sure that your exploits work within the provided VM.

Start early! Theoretical knowledge of exploits does not readily translate into the ability to write working exploits. target1 is relatively simple and the other problems are quite challenging.

GDB Tips

Notice the `disassemble` and `stepi` commands.

You may find the `x` command useful to examine memory (and the different ways you can print the contents such as `/a` `/i` after `x`). The `info register` command is helpful in printing out the contents of registers such as `ebp` and `esp`.

A useful way to run gdb is to use the `-e` and `-s` command line flags; for example, the command `gdb -e sploit3 -s /tmp/target3` in the VM tells gdb to execute `sploit3` and use the symbol file in `target3`. These flags let you trace the execution of the `target3` after the sploit's memory image has been replaced with the target's through the execve system call.

When running gdb using these command line flags, you should follow the following procedure for setting breakpoints and debugging memory:

1. Tell gdb to notify you on `exec()`, by issuing the command `catch exec`
2. Run the program using (to your surprise) the command `run`. gdb will execute the sploit until the `execve` syscall, then return control to you
3. Set any breakpoints you want in the target (e.g. `break 15` sets a breakpoint at line 15)
4. Resume execution by telling gdb `continue` (or just `c`)

If you try to set breakpoints before the exec boundary, you will get a segfault.

If you wish, you can instrument the target code with arbitrary assembly using the `__asm__ ()` pseudofunction, to help with debugging. Be sure, however, that your final exploits work against the unmodified targets, since these we will use these in grading.

Warnings

Aleph One gives code that calculates addresses on the target's stack based on addresses on the exploit's stack. Addresses on the exploit's stack can change based on how the exploit is executed (working directory, arguments, environment, etc.); in our testing, we do not guarantee to execute your exploits exactly the same way bash does.

You must therefore hard-code target stack locations in your exploits. You should not use a function such as `get_sp()` in the exploits you hand in.

(In other words, during grading the exploits may be run with a different environment and different working directory than one would get by logging in as user, changing directory to `~/hw1/sploits`, and running `./sploit1`, etc.; your exploits must work even so.)

Your exploit programs should not take any command-line arguments.

Suggested readings

In Phrack, www.phrack.org

Aleph One, Smashing the Stack for Fun and Profit, Phrack 49 #14.
klog, The Frame Pointer Overwrite, Phrack 55 #08.
Bulba and Kil3r, Bypassing StackGuard and StackShield, Phrack 56 #0x05.
Silvio Cesare, Shared Library Call Redirection via ELF PLT Infection, Phrack 56 #0x07.
Michel Kaempf, Vudo - An Object Superstitiously Believed to Embody Magical Powers, Phrack 57 #0x08.
Anonymous, Once Upon a free()..., Phrack 57 #0x09.
Gera and Riq, Advances in Format String Exploiting, Phrack 59 #0x04.
blexim, Basic Integer Overflows, Phrack 60 #0x10.

Others (Especially the first three!)

Smashing The Stack For Fun And Profit, Aleph One
Basic Integer Overflows, Blexim
Exploiting Format String Vulnerabilities, Scut, Team Teso
Low-level Software Security by Example, U. Erlingsson, Y. Younan, and F. Piessens
The Ethical Hacker's Handbook, Ch 11: Basic Linux Exploits, A. Harper et al.

Deliverables

You will have two deliverables.

1. You will need to submit the source code for your exploits (sploit1 through sploit3 and sploit4), along with any files (Makefile, shellcode.h) necessary for building them.

2. Along with each exploit, include a text file (sploit1.txt, sploit2.txt, and so on). In this text file, explain how your exploit works: what the bug is in the corresponding target, how you exploit it, and where the various constants in your exploit come from.

We require you to submit your deliverables in a certain format. **Here is the format and what you need to do:**

1. Put all your files (i.e. `sploit#.c`, `sploit#.txt`, `Makefile`, `shellcode.h` - **# represents a number between 1 and 4**) in a directory called sploits/

2. Package them into a tarball using the following command:

```
tar -cf <YourFullName>_<your SCIPER>.tar sploits/*
```

So if your name is Roy Anthony Hargrove and your SCIPER number is 101669, then the file you submit should be `RoyAnthonyHargrove_101669.tar`. **Notice the name is camel case and there is an underscore between the name and the SCIPER.** As a sanity check, make sure that unpacking the tarball generates a directory named `sploits/` with your files. You can use `tar -xvf <YourFullName>_<your SCIPER>.tar` to extract files.

If you submit a file that does not follow the format described above, we will take away some of your points for this exercise! Therefore, please make sure that you follow the format!

Grading

First 3 target programs are compulsory! Fourth target is a bonus. If you manage to get the root shell in target 4, send your code and the text file with the explanation of your solution to com402@groupes.epfl.ch. **Only the first 20 correct solutions are going to get bonus points!** FYI: for this exercise, you will not be collecting tokens!

There will not normally be partial credit, but we may make an exception depending on your explanatory writeup. We may also ask you to explain to us how and why each exploit works.

Acknowledgements

Exercise 5 is based in part on materials from Prof. Hovav Shacham at UC San Diego, Prof. Dan Boneh at Stanford and Adam Everspaugh at UW Madison. Thanks for their hard work.

Good luck!