Homework 5

COM402 - Information Security and Privacy 2018

- **This homework will most likely take longer than other assignments, so please start early to make yourselves familiar with the exercises.**
- The homework is due **Sunday, May 20, 2018 at 23h55 on Moodle**. Submission instructions are on Moodle. Submissions sent after the deadline **WILL NOT** be graded.
- In the event that you find vulnerabilities, you are welcome to disclose them to us (can even have a bonus !)
- Do not forget to submit your source files to Moodle along with your tokens.

(32p) Exercise 1: 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. **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.

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

This assignment 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.

(18p) Exercise 2: Listen Carefully

Steganography is the art of hiding data within data. Data can be in the form of files (images, video, audio etc.) or messages. The goal of this exercise is to help you explore a simple steganographic technique and understand how pieces of information can be concealed in files without attracting undue attention.

You are given an audio clip of Bryan's lecture in the WAV file format. You have to find the token in this file. The token is of the form: COM402{**<token>**}. When you submit the token, only submit the portion within the curly brackets, i.e., remove the COM402{} part.

You can find your file in the folder at:
https://drive.google.com/open?id=1fywQrzPEcRqECOqvE4Pw8xC7fhI4cLmW
Your file name will be of the form <your_email_address>.wav.
To solve this exercise, you need to look at commonly used steganographic techniques and figure out which technique has been used to hide the token.

Note: While the exercise can be completed without any packages, the wave Python module can be useful when dealing with WAV files. Link: https://docs.python.org/3/library/wave.html . Another useful tool in Steganography analysis is ExifTool - it can give you basic information about different file types.

(18p) Exercise 3: Just In Time

In this exercise, you're being asked to guess credentials on the website.

To login, you must send a POST request with a JSON body looking like:

```
{ "email": <youremail>,  "token": <yourguessedtoken> }
```

to

```
http://com402.epfl.ch/hw5/ex3
```

Don't try to brute force the token. There are much faster ways to guess the correct token.

For example, the developer here used a modified function to compare strings which express some very specific timing behavior for each valid character in the submitted token…

The response code is `500` when the token is invalid and `200` when the token is valid. Look at the body of the response, you can get some useful information too. The "real" token that you can submit to moodle will be in the response body.

This exercise will require some patience and trial-and-error, as time in networks is never 100% accurate. In order to be precise, you should calibrate your measurements first, before trying to do any guessing on the token.

(32p) Exercise 4: Is it a bird, is it a plane?

ObjectiveToday is a fictive company that offers machine-learning-as-a-service, using state-of-the-art technology. Any customer in possession of a dataset and a data classification task can upload this dataset to the service and pay it to construct a model. In turn, ObjectiveToday makes available the trained model through an API. That's really cool! In fact, even your local hospital is using ObjectiveToday's services to decide what treatment may be recommended for patients with ADHD. It's all safe in a black-box model, right? Turns

out, some smart people out there discovered a way to infer whether a given patient was part of the training set of the model, *by only querying this black-box model*. Because all patients in the training set were diagnosed with ADHD, it means an attacker with access to the black-box model can tell whether someone was a patient diagnosed with ADHD. **That's a serious privacy risk.**

Luckily, ObjectiveToday is a fictive company, and none of this happened… right? Well, not quite: recently, researchers were able to launch such attacks against Google's and Amazon's ML platforms!

To begin with, take a look at a very recent, well-written paper [1,2] that describes this attack in the wild. You'll perform a similar, albeit simpler attack, yourselves.

The setup

For this assignment, we have trained an ML model using images from the **`CIFAR-100` dataset [3]**, also described in the paper. You'll see that the link contains two datasets, `CIFAR-10` and `CIFAR-100` - you're required to use `CIFAR-100`. In brief, the `CIFAR-100` dataset contains images belonging to **100 classes**, such as train, chair, sunflower etc. In the link you'll find a full description of the dataset, as well as the layout of the images. **Please use the dataset for Python, as you'll need to write your code in Python3**. To read the data, you can use the function `load_cifar100_data()` in `utils_handouts.py`

Our ML model, which we refer to as **the target model**, is available for you to query in a docker image, in a black-box fashion: given an image, the model answers with the probabilities of the image being in each of the `100` classes. For example, given an input image image1, the black-box model replies with a vector of 100 elements `[0.1, 0.87, …, 0.05]`.

To query the target model:

- Download the docker image
  `docker pull dedis/com402_hw5_ex4`

- Start the docker image
  `docker run --rm -it -name hw5ex4 dedis/com402_hw5_ex4`

- Copy your own test image batch in docker, which contains 100 test images (more details about downloading the image batch follow in the subsection 'Goal'). **This command needs to be executed in the host.**
  `docker cp images_JohnDoe.npy hw5ex4:/target/images_JohnDoe.npy`

- Finally, query the target model with one of the images in the batch file. You'll pass to the command the path and name of the batch file, as well as the index of the image you want to query, from 0 to 99 inclusive. **This command needs to be executed in the docker container.**

```
python3 run_target.py images_JohnDoe.npy 1
```

Goal

We give you 100 images and your goal is to predict which of these are part of the training set of the black-box model. For example, we give you `[image1, image2, …, image100]` and your attack should output `[bit1, bit2, …, bit100]`, where each `bitx` value is either `1` if you believe the image was part of our training set, and `0` otherwise.

We do not expect 100% precision of your attack, as it is pretty hard / impossible to achieve :). Instead, we accept your solution if your attack makes a wrong prediction (either false positive or false negative) **`for at most 30 images of the total 100`**.

Each student has his/her own batch of test images. To download your own batch, please find at this link two files per student, which contain your e-mail in their name:

- `images_<email>.npy` - this file contains 100 test images. To read this file, please use the function `load_batch_images()` in `utils_handouts.py`.
- `labels_<email>.npy` - this file contains 100 labels (also called classes), each corresponding index-wise to the 100 test images in the previous file. To read this file, please use the function `load_batch_labels()` in `utils_handouts.py`. **These labels are useful for training your attacker models, as explained in the sections below. However, they aren't necessary for querying the docker target model.**

A very brief explanation of original attack

Please, read the paper to get a complete overview. In what follows we summarize the main steps, but for brevity we may leave out details that are important for you to carry out the attack. Please refer to the paper [1] for the details.

1. The attacker builds several **shadow models** that are similar to the **target model**, for example by asking the ML platform for a model that classifies data similar to the target model. There's no reason to believe the platform would use fundamentally different models to achieve similar goals.
2. In this step, the attacker attempts to train its **shadow models** so that they **behave similarly as the target model**. For this, the attacker trains these **shadow models** using synthetically-generated data, as follows. The attacker generates some data that fits the goal, e.g., images for CIFAR-100, and then queries the **target model**. If the model outputs a good probability for one of the classes, then the attacker selects the image as a valid data point to train one of the **shadow models**. Of course, the attacker repeats this steps many times to generate enough training data.
3. The attacker queries **each shadow model** with two image types: `type1` are images that were used for training that shadow model, and `type2` are images that weren't. It then takes `output1` and `output2`, each a vector with 100 probability elements, and labels `output1` with "`in`" and `output2` with "`out`", meaning that a particular shadow model *believes* outputs for **images in the training set** look more like

`output1` and outputs for images that were not in the training set look more like `output2`.

4. The attacker uses several **binary classifiers (also known as the attacker models), one for each class,** to finalize the attack. Specifically, the attacker feeds **all outputs** `output1` and `output2` from **all shadow models**, together with their "`in`" / "`out`" labels, to a **binary classifier** to train it. The binary classifier thus outputs "`in`" / "`out`" given an input probability vector.
5. Finally, to find out if a given image `image_test` was in the **training set of the target model**, the attacker queries the **target model** with `image_test` and obtains `output_target`, a vector of 100 probability elements. He then feeds the `output_target` vector to the **attacker model corresponding to the image's label class**, and obtains the final answer, either "`in`" or "`out`".

Why does the attack work?

You might already know an overfitted ML model is not very useful in practice, and it definitely facilitates this attack. Overfitting occurs when the model has very good accuracy for the data it was trained on, but low accuracy on data it hasn't seen. To put it differently, the model is too specific in its features. In our case, if the model is overfitted, the attacker has an easier time distinguishing between images that were used for training, and those that weren't.

Attack simplifications

Although the attack should already be possible with the information given so far and access to on-demand creation of shadow models, we want to limit the amount of time you need to spend on the exercise, and also significantly reduce the ML knowledge required for the exercise. Thus we'll make a few simplifications.

1. First, instead of fully hiding the target model, we'll give you the neural network we used: `build_shadow_model()` in `utils_handouts.py`. You can use it to create as many **shadow models** as you want. The model simply contains a few layers, which are assembled together using the **keras ML library** in Python3.

2. We also suggest that you train your **shadow models** using a random sample of images from the training set of CIFAR-100. There are 50.000 images in the training set of CIFAR-100, 500 for each class. For training your shadow models, it's enough to use **5.000 images per shadow model, as long as you make sure you have a uniform random sample over all classes**. This is important, otherwise your shadow models might be biased and thus lower your attack's accuracy. You can use more images if you want, but do not use fewer, as your shadow models may not be precise enough. And, of course, the bigger your training set, the longer it takes to train the model. Training a shadow model with 5.000 images shouldn't take longer than a few minutes. Also, **please train at least 10 shadow models** to have an attack that's accurate enough. The more shadow models you train, though, the more accurate your attack is.

3. We also give you a **binary classifier** to use as your **attacker model**: `build_attack_model()` in `utils_handouts.py`.

The items above are the simplifications we made. Besides these simplifications, the steps of the attack are still the ones described in “A very brief explanation of the attack”.

Constraints

You need to implement your attach using keras as ML library. Implementations using other ML libraries will not receive points. We recommend using keras v2.1.5 with tensorflow v1.2.1.

Submission

You need to submit a vector of 100 binary components `[bit1, bit2, …, bit100]` that represent the output for the input images `[image1, image2, …, image100]` we give you. Each student receives his / her own set of input images together with their classes (labels), as described above, and needs to make the prediction for those. As an answer, you will get either the token, or a negative answer meaning your program makes a wrong prediction for more than 30 input images.

Your submission needs to be in a json format:
`{“email”:”<your_email”,”predictions”:”[bit1,bit2,...,bit100]”}`.
Please, do not leave any spaces between the elements of the array. You need to send this json message as a POST request to http://com402.epfl.ch/hw5/ex4_predictions. You can use the `curl` command line program to send it, for example:

```
curl -X POST -H "Content-Type: application/json" -d
'{"email":"cristina.basescu@epfl.ch","predictions":"[0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]"}'
http://com402.epfl.ch/hw5/ex4_predictions
```

If your submission has fewer than 30 mispredictions, you will get the token in the response.

Resources

[1] Membership-inference attack paper:
http://www.comp.nus.edu.sg/~reza/files/Shokri-SP2017.pdf
[2] Membership-inference talk
[3] https://www.cs.toronto.edu/~kriz/cifar.html

Don’t forget to have fun and good luck !