Ira A. Fulton Schools of Engineering
Arizona State University

CSE 545: Software Security

PCTF Final Team Report

Team Name

Team0xC (pronounced “Team Moxie”)

Team Members

- Joshua Gomez, joshuago78@gmail.com
- Jonathan Chang, jachang3@asu.edu
- Michael Kotovsky, michael.kotovsky@intel.com
- Jonathan Ong, jong16@asu.edu
- Kumar Raj, kraj6@asu.edu
- Mehran Tajbakhsh, mtajbakh@asu.edu

Part 1 - Project Description and Functionality

How does your team’s project function?

I. The Initial Strategy (CLAMP)

Our initial project proposal was inspired by the Reflector assignment from week 2. We liked the idea of defending ourselves from attacks while also using our adversaries’ attacks against all the other teams. However, since all traffic would be coming from the same IP gateway, there would be no way to determine if a request was coming from an adversary or from the admin bot. A simple reflector would block the ability of the admin bot to plant new flags and would make our system appear unavailable, and thus cause us to lose a lot of points. So, we pivoted to a strategy of mimicking our adversaries instead of simply reflecting their attacks. This strategy coalesced into *CLAMP: the CTF Logger, Analyzer, Mimicker, and Patcher* (https://github.com/team0xC/clamp).

The basic ideas are actually quite simple, though the implementation was not. First, we would set up a system (*the Logger*) that would capture all traffic on the ports designated for the CTF services. The traffic would be written to text files that we could look through to identify patterns of attacks. This analysis would be done in an ad hoc manner by team members during the competition.

However, we also had ambitions of building an automated system (*the Analyzer*) that would look for outbound traffic in the logs that contained flags, go backwards in the logs and find the request (or series of requests) that resulted in the response and record them as possible vulnerabilities in a central database. A third system (*the Mimicker*) would review the recorded vulnerabilities in the database, identify the pattern of attack, and map that into an exploit script template to produce a runnable exploit. It would then update the vulnerability in the database as being “weaponized” and record the new script in a second table of exploits. An orchestration component (*the Executor*) would query the database for exploits every round, run the scripts against every opponent, record which scripts resulted in captured flags and record their successes in the database. On successive rounds, it would run new, untested exploits first, followed by the other exploits in order of decreasing past successes.

The final component (*the Patcher*) would also look at the table of vulnerabilities in the database, determine what action needed to be taken to prevent the attack from being successful, make the necessary patches to the services, and record them as being “patched” in the central database. This seemed impossible to automate in 3 months, or even 3 years, let alone the 3 weeks that we had before the PCTF. Therefore, we decided to prepare a set of checklists that could help the humans automate their own thought processes while patching. One checklist would be used for the initial server setup; a kind of “hardening” of the services. The second would be used for patching the code of the services. It would contain a list of obvious vulnerabilities to be on the look for (buffer overflows, SQL injection, etc.), and a complimentary list of strategies for mitigating them.

II. The Revised Strategy

We knew from the beginning that the function of the Patcher would really be done by the members of the team, not by an automated process. However, given the short time frame, development of the Analyzer and Mimicker also seemed like unreachable goals. This was unfortunate, because it meant the only automated parts of our strategy would be the running of the attack scripts and traffic capture. We had no automated defenses, even though that was the heart of our original strategy. To compensate, we came up with two new defense strategies that, while not dynamic like a reflector, were easier to develop in a short time frame and were likely to be highly effective.

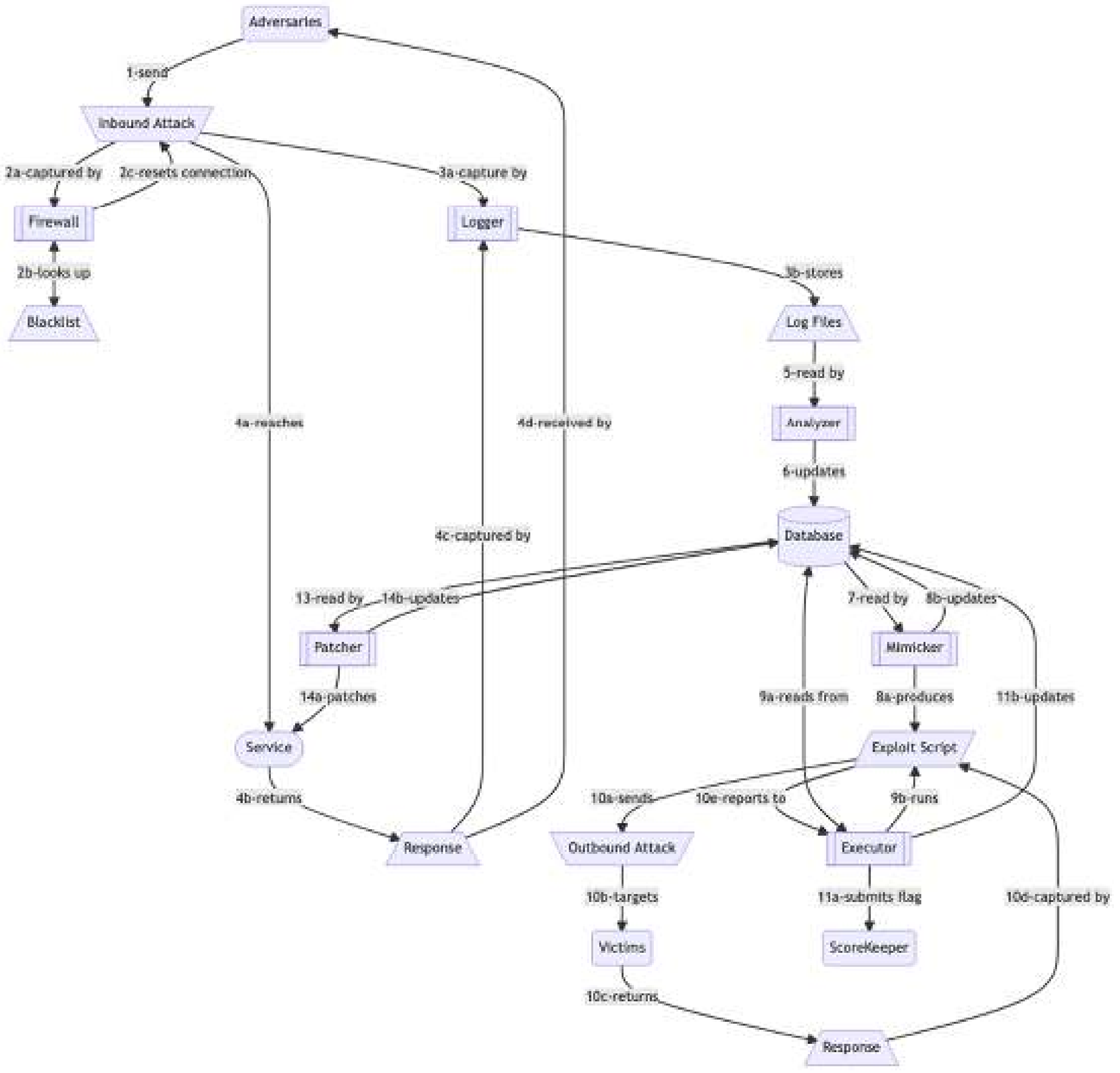
The first addition to our strategy was *the Firewall*. This system would capture all inbound traffic on the service ports (similar to the Logger). It would compare the contents of the requests with a list of forbidden values, based on well-known exploits, such as “../” for directory traversals, “cat” and “bin/sh” for command injections, and so on. The Firewall would not intercept traffic, it would only capture it. Thus to play defense, it would reset the TCP connections of those requests before the vulnerable service could respond. To ensure our Firewall could do this, it would configure a network delay of 1000 milliseconds for the entire server.

The second addition to our strategy was a collection of encryption and decryption code snippets. We wanted a way to make stolen flags unusable for our opponents. Encrypting the contents of the flag files would prevent opponents from earning points because the flag would be incorrect. However, simply encrypting the files would also make us lose points, as the admin bot would fail to find the flag it planted when checking the service’s availability. Our solution would need to decrypt the files for legitimate requests. Thus, instead of performing encryption from outside the service, we would patch the services to use the encryptor/decryptor snippet when creating new content and reading existing content. Any reads done within the service would thus result in a decrypted file. Reads done outside the normal control flow (such as through command injection) would result in an encrypted version of the file. Our strategy was to create an encryption/decryption code snippet in each language we expected to see in the CTF (C, Python, PHP), so that we could quickly plug it into any reads or writes done by the services.

Our project revision also involved dropping the central database. Although it was the first part of the codebase to be developed, we ultimately deemed it to be an over-optimization. Since the vulnerability analysis (Analyzer) and attack mimicking (Mimicker) would be done ad hoc by the team members and not by automated systems, there was no need to record vulnerabilities in the database. And we assumed that we would only have a handful of exploit scripts running, so sorting them based on past success was unnecessary.

III. Technical Details

Here is a high-level view of the overall flow of the system. Note that, in the revised form of the project, the functions of the Analyzer, Mimicker, and Patcher are performed by human members of the team, not automated processes.



A. The Logger

Packet capture is already a deeply explored topic, so rather than reinvent the wheel the goal was to pick a technology that would minimize system impact while maximizing compatibility and stability. To this end, dumpcap, as part of the Wireshark package was utilized to do the capture as a fully featured lightweight client that could not only capture pcap files and work with standardized filters, but also carried the least overhead as compared to other products with the required feature sets. Beyond that, system resource management was of the utmost priority. You no sooner want your disk filled with captured datafiles than you want memory/cpu being wasted inefficiently.

Since we needed rapid deployment, ease of use, and reliability, an install script takes care of all the nitty gritties:

- Install all system dependencies via apt;
- Setup absolute paths for required service scripts;
- Setup a custom system service for ease of control with standard service/systemctl commands;
- Setup crontabs for system maintenance;
- Setup users/groups/permissions;
- Include administrative utilities.
- NOTE: an uninstaller also cleans up most (accepting users/folders/groups) of these features.

All of this was designed to be easily configurable in one location, providing options for:

- Directories for capturing/archival
- Names for users, daemons
- Standard and ease of use filters for dumpcap
- Disk compression configuration
- Archival parameters:
    - Time until compression
    - Retention
    - Maximum disk utilization
    - Optional secondary archival location/retention/disk utilization for long term storage

The cron jobs (defaulting to 1/min) updated ownership/permissions, managed disk utilization and recycled services automatically upon configuration changes to keep ease of use and utility high when you optimally want to be focusing efforts on the competition, not the maintenance of your tools.

This portion ended up being fully implemented and achieved satisfactory live results. Barring one permission snafu on the parent folders, this was deployed quickly, operated unobtrusively through most of the event and helped to discover both potential exploits and patches for the majority of services in the competition. Analysis was done manually via Wireshark after using scp to remotely retrieve logs and filtering on the service in question.

B. The Firewall

We anticipated some of the services in the final PCTF challenge will be exploited by injecting shellcode which led us to the development of a firewall for our host VM. The firewall designed for the project carefully analyzes incoming TCP traffic data and based on pre-established rules reset the TCP connections which look malicious.

Firewall also acts as a TCP data monitor and generates recurrent summaries at defined intervals for manual analysis. These summaries were used to refine our blacklisting rules which further improved protection of our services.

Firewall consist of following three modules:

- Data Capturer: Captures the incoming TCP traffic data, decodes it and queues the messages in a processing queue maintained in Redis
- TCP Kill Workers: For scalability multiple workers running as separate processes keeps dequeuing the messages from the processing queue and check for existence of blacklisted keywords in the captured data and if found uses a TCPKILL utility to reset the established TCP connection from the malicious source IP and Port. It exit after queuing the message further to the summarizer queue
- Summarizer: Accumulated messages in the summarizer queue are processed for generating aggregated summaries at defined intervals.These aggregated summaries assists in manual analysis and refining of the firewall rules.

C. The Encryption Snippets

The idea for encryption was brainstormed when trying to find alternative approaches based on the principles of the CIA triad and adversarial mindset, but not directly utilizing methods taught in this class, which were likely to be used and expected by other teams. Conceptually, it also seemed complementary to the analyzer as there would be minimal interference, and it required little to no foreknowledge of specific vulnerabilities and could theoretically be patched immediately. The criteria for the encryption function was based on an analysis of the specific context of the CTF event. It was expected that the script bot would always provide a username and password, so both should be used to authenticate the user as the script bot. It was also expected that the server would never need to know the username, password, or file contents, except when provided valid login credentials, so they should never be stored as plain text or decryptable by the server without the login credentials. It was also expected that an attacker would know a username, but never a password or flag, so encrypting the username would make it so the attacker would be unable to identify a file associated with the username. Encrypting the file contents meant that flags could not be bulk submitted even if they were all collected. Encrypting the password meant that the attacker could not log in using the known flagID and encrypted password. Because the event only lasts 24 hours, and each flag expires in approximately 15 minutes, the algorithm only needs to resist reverse engineering for less than 24 hours, and resist individual cracking attempts for 15 minutes. Adjusting for the opportunity cost of cracking encryption unique to one team, it is further unlikely that any individual team would devote the time on an exploit that only works on a single team. Given the relatively low cryptographic requirements to maintain adequate confidentiality, priority was placed on developing an algorithm that prioritized integrity and availability, or rather, an easy to patch function that prioritizes compatibility with an unknown program while not compromising preexisting integrity and availability.

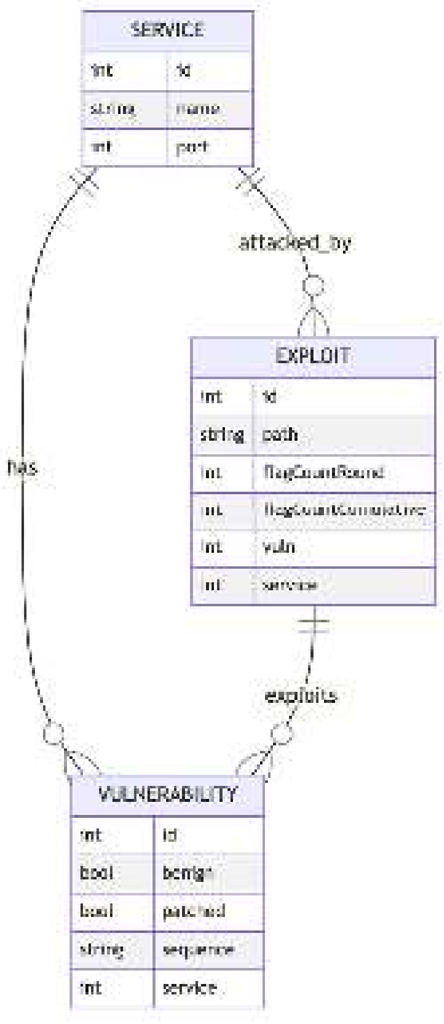
Encryption code snippets for patching the services were written based on the following design criteria. The encrypted string should share several characteristics with the unencrypted string in order to maximize compatibility and minimize the need to make changes to the original program, such that the encrypted string follows the same pattern as the unencrypted string. This meant having alphanumeric characters in the same positions with non-alphanumeric characters never added, non-alphanumeric characters in the same position unmodified in case the program uses them in some way, and identical string length. It should also be able to use an arbitrary string as a key. The code should require an absolute minimum of external libraries, be minimal in size, be only a single function to encrypt and decrypt, and be easy to translate between languages in case services reference common files. Cryptographically, it should require a user provided key to encrypt or decrypt any information, store no sensitive or identifying data as plaintext, be sufficiently difficult to crack even if the algorithm is known, be difficult to detect a pattern (no obviously ciphered “FLG”), and have a domino effect where a single character change in the string or the key causes all following characters to change.

Standard hashing and encoding methods were evaluated and it was determined that they may not satisfy these criteria. For example, ROT13 is involutory, but alphabetic only and trivially easy to detect and decode since the “FLG” prefix will always be stored as “SYT.” Bitwise XOR masking presented the problem of causing alphanumeric characters to potentially map to non-alphanumeric characters and would have been a substitution cipher that also would have been easy to detect. Other encodings such as base64 have the same prefix detectability and easy decoding issues with the additional problem of being longer than the original string, as well as possibly containing non-alphanumeric symbols, which could cause problems in the original program. A hash function like SHA1 could be used to hash usernames and passwords, but it would still be possible to identify the hashed flagID. Salted hashes would solve the issue, but do not solve the issue that hash outputs are usually not strings of equivalent length and hashes do not preserve the integrity of data meant to be retrieved and not merely verified. Therefore it was decided to proceed with coding a custom encryption function.

Details for the actual encryption algorithm can be found in the documentation in the codebase. It is a rotor based design that incorporates features from the Caesar cipher, enigma machine, and Dial-A-Pirate and primarily utilizes integer addition, modulo, and string/array indexing which are readily available in almost all programming languages. Caesar cipher, used by famous dictator Gaius Julius Caesar, was used to simplify rotor encoding because shifts could easily be implemented using only addition and modulo. Certain aspects of enigma such as the advancing rotors were useful in encrypting the “FLG” prefix without easily detectable patterns, and the reflector makes it useful for both encrypting and decrypting. The design is also relatively portable compared to the Lorenz cipher, which translates into simpler code portability. Even though Lorenz is more secure and required the first electrical digital computer to crack, the Colossus ran at 0.0058GHz, and enigma was vexing enough that cracking it was a major undertaking during WWII and computer scientist Alan Turing contributed the design for the electromechanical “bombe” computer to crack it will full knowledge of the enigma algorithm. *The Secret of Monkey Island* was many guys' first brush with anti-piracy digital rights management. The included “Dial-A-Pirate” cipher disk, while not the first of its kind, was meant to help stop pirates attempting to discover the secret of Monkey Island™. Prior art had often been static manual passwords, but cipher disks made the key a disk based decryption device. Using the key string here as secondary rotors helped restore complexity to the encryption that had been lost with the simplification enigma to using a Caesar based rotor. Functionally it is utilized the same as any other encryption/decryption method that would satisfy the aforementioned criteria.

D. The Database (dropped)

The Database was designed to serve two purposes: 1) tracking the identification and patching of vulnerabilities in our services, and 2) tracking the success of our exploit scripts. The schema had one table for each of those concepts, along with a table for the services themselves.



Each service would get an entry in the Services table. The only useful information in this table is the port. Aside from that, this table is mostly used as a method for grouping rows from the other two tables based on their relation to the services.

The Vulnerability table would be updated as the analyzer (human or computer) found problems with the services. If a sequence of requests and responses had been found in the logs, they would be added to the “sequence” field. If the vulnerability was a false alarm, the “benign” flag would be set to true. If it was real, then we would patch the vulnerability in our own service and exploit it against others. Once it was patched, the “patched” flag would be updated to true.

When an exploit was written, a row would be added in the Exploit table, and it would refer to both the Service it attacks as well as the vulnerability that it takes advantage of. The Vulnerability table also contains counters for the number of flags captured by the exploit in the previous round, as well as a counter for the cumulative number of flags it had successfully captured. The Executor script would use those numbers to rank them by “most successful first”. It would also use the “path” field to find the script file to run.

E. Executor and Interceptor Scripting System (unused)

This system consists of three services that run independently, using event-based architecture on a main loop: executor (for attack scripts), interceptor (to reset offending TCP sessions), and a fake flag setter. First, the fake flag setter monitors each of the flag directories under either ‘<service_name>/append’ or ‘<service_name>/rw’ for new files and sends out a boolean scripting flag if it sees any updates. Using a predefined ratio, it then logs all the new genuine files into text files, then creates a number of fake files with the same flag token but different flag ids, each containing a random number of fake flags. Users who, for example, cat using the flag token and a wildcard would see all the fake flags along with the real one. This is used internally in the system in two ways:

1. Upon new genuine files at the start of a tick, the executor assumes the script bot might have begun to set new flags, so it executes a new thread to run all its attack scripts against every team in which the swpag_client reports the scripts’ targets are up.

2. In sessions initiated by other teams, if fake flags are detected by a scapy sniffer in the response packets being sent back to them, the interceptor assumes the real flag is among them and spams a random amount of fake flags in TCP packets while simultaneously setting the reset flag.

This is made possible by a main event loop that runs on an infinite while loop, using time deltas to periodically query the swpag_client for tick info. It uses this info to set up the following events: the start of a new tick, halfway through a tick, the last ten seconds of a tick, constantly (every half second), and every five seconds. After being initialized, the various methods of each service are executed, sometimes conditionally, in one of these events. For example, executor scripting executes on a new thread upon the first flag update per each tick. The scripts are reloaded two times per tick: halfway through it, and near the end. The interceptor checks for new genuine flags to determine which patterns to exclude when the sniffer callback consumes them to detect fake flags.

In more detail, both the executor and interceptor dynamically reloads scripts two times per tick. This allows for new scripts to be added without restarting the entire service. They load the script modules into memory; then, when the scripts are ready to be run, they execute according to a predefined template that specifies its target service. The executor queries swpag_client for target services, then for each team that has the service still up, it executes the respective scripts. The scripts return a list of flags if successful. The flags are batched into 100 flags per submission to avoid the possibility of ‘too_many_incorrect’ problems. The number of correct flags is then returned so that these exploit scripts could be sorted by most successful, so that in the next tick, the most effective scripts are run first.

The interceptor, on the other hand, starts a new thread to sniff at the beginning, but the event loop periodically checks to see if it has failed to respawn, perhaps due to bugs encountered while interacting with other teams. It has two default scripts that are applied against sessions initiated by other teams to all services: fake flag detection and timeouts. Fake flags have already been discussed. If 10 seconds have elapsed between TCP requests or if the entire session takes more than the time of a tick, then the session is reset. This is mainly to handle edge cases to prevent manual exploration or scripts that iterate or fuzz. Additionally, it allows custom scripts that test for any condition given the payload and packet direction and returns whether the session should reset.

Finally, the interceptor sorts every TCP session using unique identifiers – at any single time, there will not be sessions established with duplicate target services and attack ids and ports. Therefore, the interceptor could track which packets belong to which attacker, and log them in separate text files. A format was devised to maximize usefulness. The files are stored in directories by service and tick number. The file names include the number of packets sent by the attacker, the reason code of why the session has been terminated, and is suffixed with “[FLG]” if a flag-like string has been discovered in any of the responses, since these sessions have a high likelihood of success and deserve immediate notice. The logs themselves decode the payload to ascii if there are corresponding visible characters, but otherwise maintain the bytes to make identification of nopsleds and shellcode straightforward. They also include start and end times, and a more detailed reason for termination. Please see the code repository readme for examples of logs.

One issue with such a comprehensive system was the amount of time required to test it. Stubs of the swpag_client interface were used with a timer that counts down the approximate seconds left. The services were separated into their own modules as singleton classes so that they could be invoked in other scripts and passed around, which was also helpful in testing. Procedures were moved out to their own methods whenever possible so the logic could be invoked individually. A small number of methods specifically dealt with things such as manually manipulating scapy packets. All useful methods included docstrings to help identify them at a glance.

While the initial conception included only the executor, it was realized that these services benefit in coordination. The system became a one-two punch that attacked at the first opportunity, while diminishing other teams' opportunities to strike back. It was ultimately unused because it was over-engineered for the task, relied on some faulty assumptions (such as that the script bot simultaneously sets flags at the start of every tick) that required revising during the competition, and that Kumar got his firewall up and running faster. His had less moving parts, so we deemed it more reliable. And in order to maximize point acquisition, we decided to focus on running ad hoc scripts as soon as we can rather than rescripting the backend to deal with changes.

F. Checklists (unused)

Multiple checklists were drafted to cover points and features discussed in our meetings for the development of tools, to cover activities that should be completed in the first hour, and one to act as a reminder of tasks to perform while participating in the event. However, it ended up that many tasks relating to specific tools were performed by individual team members and not the entire team as a whole.

G. The Analyzer (unused)

The Analyzer module is written in Python and it works with the Vulnerability table. It is responsible for: 1) finding the flags in captured traffic that are stored in the PCAP files by the logger. 2) finding any anomaly pattern in the captured traffic and adding them into the Vulnerability table.

The Analyzer module uses Python Scapy library to analyze and manipulate the captured traffic. Captured traffic is read from the PCAP files by the sniff() function. Then in the first step, the program searches it for flag patterns (define regex expression as 'FLG[0-9A-Za-z]{13}'). If any match is found, the Analyzer module tries to submit the found flag to the scoreboard to get points. Then the Analyzer module searches for anomaly patterns (Prepare a list of vulnerable strings in different categories: Network Functions, Backdoor Functions, System Functions, Process Functions). If a match is found then the Analyzer uses rdpcap() function to find all the traffic streams related to the session containing the anomaly. It then finds the source of the traffic and other useful information and builds them into a string and searches for any duplication in the Vulnerability table. If our string is novel, then it adds a new record in the Vulnerability table and populates the sequence field with our string and marks the benign field as False.

Part 2 - Project Performance Expectations

How and why did your team expect your team's project to help you succeed in the PCTF?

Firewall

We knew that it would take a significant amount of time to analyze the services and discover their vulnerabilities. We also suspected that some teams might attempt a "shotgun" method of exploitation (i.e. blindly sending out random data or common exploits for SQL or command injection) from the very beginning of the competition before anyone began patching their services. We also knew that some teams may just be faster than us at performing the analysis. We built the Firewall with an expectation that it would provide a shield for our vulnerable services to protect them from these early attacks in the first few minutes or hours of the competition, while we were still performing our own vulnerability analyses on them.

Encryptors/Decryptors

We knew there was a good chance that we would not discover all the possible vulnerabilities and that an attacker may be able to read data from our file system, including flag files. We built the encryption/decryption code snippets with the expectation that they would provide a second level of defense. By encrypting all data written by the services, we ensured that anyone who accessed the file outside the normal control flow of the service would be unable to identify specific target files, and would end up with file contents that would not result in points. They would be unable to find the file, or read files to get flags but would not get flags. If they did not bother to check, and submitted the contents to the game system, they would fail to get any points. By decrypting files that were read within the control flow of the service, we would not lose any points when the admin bot performed its accessibility check.

Logger

The heart of our initial strategy for CLAMP was based on mimicking other teams' successful attacks against us. To perform such mimicking, we needed a way to see what attacks were successful against us. We built the Logger with the expectation that it would provide an easy way for us to look through traffic logs and see the contents of all requests and responses, and therefore easily identify which responses contained flags, and what requests resulted in those responses. One team member could then quickly write an exploit script to perform the same attack against others while another member patched our own service to mitigate the attack.

Executor

We knew that each round was approximately 3 minutes, and that our scripts would need to be run against all opponents on every tick. We built the Executor with the expectation that it would automate the running of our exploit scripts. We would write a script, put it in a specified “exploits” directory, and the Executor would fire them off every time it was triggered by a new tick.

Patching Checklists

We conceived of the checklists with an expectation that they would help us patch the services in a calm and thorough manner. We knew it would be difficult to spot some of the vulnerabilities, and that it might be difficult to really focus on the code and recall all the things we have learned in the past 6 weeks while so much chatter is happening between teammates. By preparing the checklists ahead of the competition, we could help ourselves during the game and prevent us from forgetting to check for certain types of vulnerabilities.

Part 3 - Project and PCTF

How did your team’s project help or hinder your team’s performance in the PCTF?

What Worked Well

Our project was devised to be a cohesive set of tools that employ both short and long term strategies, expediently responding to the situation as the game progresses. While most of our efforts bore fruit, some of our underlying assumptions of the structure of the game were challenged. Our defensive tools were especially advantageous: our firewall helped secure an early advantage while our encryptor scripts paved a path to patching a service later on. Our easily configurable logger was able to capture exactly the data we needed to defend against other attacks, as well as copy them against other teams. However, we opted to utilize ad hoc attack scripts rather than our executor system to take advantage of speed and flexibility.

We knew from the outset that the challenges in the competition, whether developing attacks or patches against attacks, take time. After recognizing a vulnerability, a team would have to figure out the best library and script the process to reach it, as well as quickly test for the differing string formats of the flag and token ID. Therefore, Kumar's firewall quickly blocked the appearance of certain keywords immediately after identifying the vulnerabilities of the respective services by sending a reset flag against the offending TCP session. We knew that, in order to allow patching against exploits, the script bot would only test against the correct flag specifically, and therefore any command or language that casts wide nets, such as using 'cat' over a directory, or wildcards like '*', would be indicative of an attack. The firewall allowed us to quickly hinder these simple attacks while we patched them since it was easier to simply specify keywords than to script a sequence of commands. We observed through the logs created by the firewall each time it blocked a session that it had successfully prevented many attacks, especially in the early stage when services were unpatched. It helped immensely even up to the first 16 hours.

While the firewall provided necessary short-term defense, our encryptor scripts reduced our response time in patching services that relied on C or Python code. It was Jonathan Ong's brilliant idea to create a backend for the services that encrypt and decrypt their outputs to flag files. The script bot, which knows the correct flag, would be able to query it through the service, which properly decrypts the flag. However, any exploits that reach the flag through a backdoor would only see the encrypted files. Jonathan Ong expertly identified the opportunity to adapt these scripts for the Backup (1) service. To our knowledge, by analyzing our logs, no attacks succeeded against that service since this strategy was employed.

The aforementioned tools provided defenses when we had an idea of what to target. On the other hand, Michael's configurable logger developed our intuition when we didn't. The logger had the capacity to focus on particular services and ports so that we were able to hone in on what we were looking for. For one, we were able to achieve a basic understanding of the reliability of our patches by observing the incidence of flags being sent back in our responses against potential attacks. But the highlight of this tool shined in two specific instances. First, rather early off, we detected a dot-dot attack against our Say What (2) service, which we quickly reproduced in an attack script against other teams. Second, immediately upon noticing team Young-Grasshoppers exploiting the final service, Sampleak (4), we pored through our logs and found their attack, including their exact shellcode, which we turned back against other teams in our own script within an hour. Of course, Mehran was on the brink of solving this problem, so using the general outline of his script helped.

What Went Unused

The Executor was one tool that we did not employ, due to some early realizations. In the hour preceding the official start of the competition, Jonathan Chang nearly finalized an attack script against the Backup (1) service, and ran it ad hoc (without the executor system) to gain early points. We previously assumed from reading the ICTF framework Github that a "setflag" script would be run at the immediate start of a round, followed by any number of "getflag" scripts to test for its existence. We reset the script a number of times to hopefully time it to run after the "setflag" script so that we could get the flags as soon as the round started, in order to give the scripts plenty of time to run. Within a couple dozen ticks, we gave up on that notion, since the flags were set anywhere up to 90 seconds in a 3.5 minute tick. Rather than fixing the backend to deploy these scripts, we made the executive decision to run attack scripts independently as background processes so that we could focus on developing exploits instead of administrative tasks.

Each attack ran on an infinite loop. They were synchronized to game ticks, but ran multiple times per round regardless to increase chances that we obtain the flag soon after the script bot sets it. Since the timing of the “setflag” script is unpredictable, there would have been the chance that a round was skipped if we played the attack just once at a determinate time each round. It turned out that the boilerplate in checking for the start of a tick using the swpag_client was so minimal, that using a coordinated system seemed more capable in theory than in practice. In practice, we only developed one attack script per service in a small number of iterations, so a script launcher would have been overkill.

The checklists also went unused, though not because they lacked value. This was simply a matter of the team being completely inexperienced at playing in a CTF and forgetting to make use of them. When we first logged in we focused on getting our github repo installed. Then, out of pure curiosity, we simply began poring over the code of the services independently, without a plan for who would look at which service or who would perform patching versus writing exploits. We relied on the knowledge we gained from this class to identify some of the early command injection exploits for Backup (Service 1) and a SQL injection attack on Flaskids (Service 3). While patching the Python service we inadvertently brought it down and spent a lot of time trying to figure out what went wrong. Eventually we discovered that the wsgi service needed to be restarted in order to pick up the changes in the Python code. The somewhat chaotic flurry of activity in that initial couple of hours caused the team to forget about using the checklists. The bit of extra preparation by consulting with the checklists could have provided better team organization and prioritization for a more seamless experience.

How could your team improve your project to perform differently in the PCTF?

We were very satisfied with the three main components: the Logger, the Firewall, and the Encryption Snippets. If given more time to prepare, some possible refinements would include:

1. *Logs that are (more) human readable*: As mentioned above, to analyze the logs the team had to retrieve the files off the server and load them into Wireshark locally. Although this was a perfectly fine setup and worked well for us, it might have decreased analysis time if the logs were sorted into different files, each marking a separate TCP session from a different service. One possible refinement would be to make the captured logs immediately analyzable by the team members directly from the game VM by using some transformation scripts to make them into a more readable format of text file. If we had enough time, a visualization web service accessible through a browser could provide easily filterable updates in real time. We have not put much thought into what *could* be done in this realm since we knew it was not within a realistic scope for the project. But the possibility of going further is certainly there.
2. *Exploit metrics:* We abandoned the Executor during the game because independent scripts running on infinite loops were faster to develop and more flexible; therefore, they worked better than an orchestrator attempting to synchronize itself with the game ticks. Similarly, we abandoned the database as an over-optimization. However, there is a nugget of usefulness in their core ideas that could be used to refine our attack strategy. It could have been useful to track which teams were successfully defending our attacks, so that we could look through the logs and analyze their responses to update our attacks. It would also have been useful if a log kept track of the length of time in attacking each team so that we could attack the easiest ones first. Our Flaskids (Service 3) script continually hung against a particular team, which we discovered way too late. If we had that data stored in a single place, whether it be a database or just a flat file, it would have been easier for the entire team to respond to problems as they arose and investigate potential fixes, whether they were due to our opponents patching their services or some other problem.
3. *Drills and Game Plans*: The checklists were a great idea that we did not make use of. Our defense could have been more thorough and efficient if we had been better about using them. In his book, *The Checklist Manifesto*

[3], Atul Gawande explains how checklists can literally save lives and limbs, whether it's airline pilots performing pre-flight diagnostics or surgeons performing pre-operation checks that they should be amputating the right leg and not the left leg. But, he also demonstrates that checklists only work when teams adhere to them with a near militaristic discipline. If given the chance to play the CTF a second time, we could pitch the checklist as a central document to plan all aspects of the project, in addition to several practice runs in which we perform our setup on the VM, so the whole team would have experience using the checklists first. A central document like a checklist would also allow team members to report what they're working on and provide status updates, so that we could find distinct tasks more efficiently, without overlap, or offer help when it would be needed.

4. *Encryption patching*: Even though the encryption was sufficiently secure, it could have been implemented in a more secure way. Although this was not an issue in the CTF event because the source code was not available to attackers, there was a vulnerability in the case that an attacker got access to the encryption algorithm, all encrypted usernames, all encrypted passwords, and all encrypted file contents. The attacker could attempt to encrypt/decrypt all passwords using the single known unencrypted username, then attempt to decrypt the associated file contents to search for the "FLG" prefix. The solution is to not use the username as a key, since it is a given that the attacker will never know the password or the flag. Other approaches would have been to use the file contents as a key, but this would have necessitated a change in program logic, encrypting the password with itself as the key and hoping that it would be cryptographically complex enough to resist cracking for the required timespans, or using only the encrypted username, which is a function of both the unencrypted username and password, for validation and not recording the password at all, encrypted or unencrypted. False flags could also be planted for easier exploit detection using loggers.
5. *Obfuscating exploits*: It is unknown how many other teams used logging and/or firewalls, but there seems to be a number based on the pattern of exploits being found by one team then quickly duplicated by others. Developing a library to quickly obfuscate attacks and also make meaningless non-exploit requests to flood logs would help delay or prevent duplication by other teams, and also allow us to bypass blacklists. Any number of methods for encoding could be used. For example, the command `rev < <($(python -c'print"\x2a tac"[::-1]'))` instead of `cat *` may have bypassed detection methods that would have searched for `FLG`, `cat`, `*` or `|` in the request and the response packets. There are several other encoding schemes that could be used as well. Combined with log flooding to make manual review impractical, it would be excessively difficult to filter through log entries without knowing what to look for.

PCTF Observations

- Service 1 - Backup
  - Exploits
    - A command injection was found after reading through the source code.
  - Patches
    - Quick firewall implementation was able to deny most non-automated attempts to steal our flag early on and surprisingly remained effective throughout most of the event.
    - In order to effectively block command line injections from getting anything useful, a custom encryption technique was implemented to obfuscate data that wasn't directly read by the program. They could directly search all they wanted but with file names that didn't match their flags and garbled file contents, it wouldn't be worth anyone's time to break it for just our server.
  - Observations

        - `system(cmd)` was a major vulnerability that did not need to exist and could be replaced with a file read and `puts()`.
        - It did not seem to ensure uniqueness of usernames.
- Service 2 - Say What
    - Exploits
        - A dot-dot vulnerability that allowed access to the ‘append’ directory by using the ‘path’ variable on the query string was found in the logs and replayed against other teams.
    - Patches
        - The php was patched to use basename() in order to avoid directory traversal attacks
        - The initialization code was disabled since it wasn’t needed by the service and it was safer to not suffer the same exploits we were utilizing.
    - Observations
        -
- Service 3 - Flaskids
    - Exploits
        - A sql injection attack was utilized to leak additional information about flags.
        - The default password was utilized to trick other teams into re-initialization and caused service outages for 4-12 teams on average until they figured out how to patch this one. This was launched after our directory traversal so we could get flags before resetting it and denying other teams the availability to do so.
    - Patches
        - SQL was parameterized to prevent the same injection attacks
        - TRIGGER ON BEFORE DELETE directives were added to sqlite databases to shore up any injection vectors we missed.
    - Observations
        -
- Service 4 - Sampleak
    - Exploits:
        - Minutes after first blood was announced we had an attack signature found in the logs. We’d been banging our head against this buffer overflow for hours and had a baseline understanding, but with working shellcode and a couple successful attacks identified in the logs, we were able to recreate and replay to recoup significant points early in the AM.
    - Patches
        - Shortly after employing our attack, the string “%u %200s\0” was found in the strings table using a hex editor and overwritten to “%u %58s\0\0”
    - Observations
        - The one vulnerability identified was found in the read segment, a buffer of size 60 with a following read statement that read 200 characters.
        - Offset from”execution id” was essential for determining the proper memory address.

Part 4 - Peer Contributions, Collaboration, and Group Participation

How did each member on your team contribute to specific parts of the project and experience?

It's undoubtedly true that a project's end product is the most visible achievement, but a great amount of other work helps bring it to fruition. Work that goes into leading, coordinating, planning, documenting, as well as the parts of code that did not end up being used, all help refine the end product. They should not be discounted or trivialized. The following list of contributions shall be enumerated in two categories: the effort spent on the project before the CTF competition, and the effort spent during the competition. All members participated in weekly Zoom meetings where we contributed our ideas and concerns.

I. Project Contributions

Joshua Gomez (Team Captain)

Joshua led the organization and planning efforts of the team. He created the private Slack group, coordinated team communication, scheduled meetings, and created tickets in the GitHub Project. He also took input from the team and formulated the CLAMP strategy, which was a synthesis of everyone's ideas. He then created the group's Github code repository and wrote the project proposal documents.

Joshua's technical contributions to the project were focused on the SQL database. He designed the initial schema, created ORM models using SQLAlchemy, and validated them with fixture data and unit tests.

Jonathan Chang

Jonathan was assigned to implement the executor based on initial interest, but it was Jonathan Ong that actually wrote the initial outline. Jonathan Chang then organized the code to run it in a main event loop, launch scripts in new threads, and allow more customizability in launching the exploit scripts. However, due to lack of flexibility and speediness in development, since the scripts needed to be contorted to a certain format to be automated, the executor was never used.

Jonathan also developed an alternate firewall system that sorted TCP communication by source port in order to better organize logging, and generate fake flag files to be used as a honeypot to reset TCP sessions if they were discovered in the responses. These efforts were also unused in the competition due to overlap, since Kumar's firewall was provably reliable. The short timeline and lack of agreed upon API made it hard to rely on each others' parts as a dependency. So there was sometimes a great deal of overlap.

Michael Kotovsky

Due to his prior expertise, Michael was assigned to implement the logger. His code base in shell script is among the most expansive in the project and among the first to be fully committed to the repository, which helped anchor our expectations. Project development started with a fully detailed scope, deliverables and methods to generate sample files (as to not gate development); this continued with research and testing of appropriate existing products to utilize as well as copious documentation within the scripts, configurations and project readmes. Further feature improvements included adding additional usability, functionality and scope beyond the requirements to make for a seamless and convenient user experience (since we would be its first user base) while stressing maximal compatibility with minimal system impact. The end product could have been left running indefinitely if adequately configured and continue to work through reboots, power issues and other service interruptions.

Jonathan Ong

Jonathan focused on writing encryption and decryption scripts, an idea he advocated for, which turned out to be warranted. Prior to the competition, he wrote a number of these scripts in different languages to maximize chance of usage. Jonathan also wrote the initial outline of the executor. Throughout the project, he kept the team on pace by documenting checklists of what needed to be done, evaluating the viability of different approaches based on what we knew of the CTF event, and what required attention at each phase of the process. Although we didn't end up using the checklist during the competition, these provided structure in our experience and were still helpful.

Kumar Raj

Kumar implemented the firewall out of interest, which was an idea he devised and convinced us of its value. It monitors TCP packets, sends reset flags upon incidence of a list of keywords, which could be later customized. It also logs the packets whenever it resets a session. Kumar also spent great effort attempting to link this firewall to Google API to improve the experience of using it, so that we could add keywords and monitor traffic over a spreadsheet. Ultimately, since connecting to a third-party might have presented connectivity issues, this idea was scrapped.

Mehran Tajbakhsh

Mehran's responsibilities in this project were mainly focused on the Analyzer module. The main job of the Analyzer was to help interpret log files and update the vulnerability table with detected anomaly patterns and flags found in the captured traffic. Developing a Python module for him as a beginner Python programmer was a big challenge for him. He accepted his part and did his best to do his responsibility to an acceptable degree. He faced another challenge since his part was dependent on other team members' parts. For the Analyzer module, he wrote a script with most functionalities (find and submit flag in the scoreboard and search for vulnerability patterns in the captured PCAP files including all the related streams). During our last meeting, our team decided to drop the database so he could not test his module before the competition. Hence, we could not use his module during the PCTF.

He used the PCTF environment as a good opportunity to test and evaluate his code with real traffic, threats and vulnerabilities, and wrote down useful notes in his code to improve and implement a more practical version of his code in the future. He also provided a number of readme's with useful resources that came in handy during the competition.

II. PCTF Contributions

Joshua Gomez (Team Captain)

During the competition Joshua focused pretty heavily on systems support and vulnerability hunting. He was the first to log into the VM and installed the project repository so that other team members could get their systems running (e.g. the Firewall and Logger). He also backed up the four services to a clean GitHub repo.

Joshua has experience with web development using Python frameworks, so he began analyzing the Flaskids service first, and quickly found the obvious "changeme" password vulnerability. He patched the service immediately. Unfortunately, the patch did not take effect, and our opponents' attacks resulted in the service coming down completely (it's been a few years since he was an active developer, so he forgot about needing to restart the wsgi service). After spending time troubleshooting, he was able to bring the system back up by restarting the container. He shared this information with the team so they could perform similar operations when he later logged off. This information sharing was valuable, as that service in particular was the most finicky and needed to be restarted multiple times throughout the competition.

Joshua also found the two-step SQL injection vulnerability in the Flaskids (3) service, and shared it with the team. Mehran and Jonathan Chang ran with it to develop the exploit script while Joshua patched our own service. After feeling confident about the state of Flaskids, Joshua turned his attention to the PHP service, Say What (2). He decoded the files and added them to the group repo so that all the team members could see the source code of the service and hunt for vulnerabilities.

Joshua participated during the first 7 hours of the competition, from the initial hour of setup at 4pm PST to 11pm PST. After getting some sleep, he logged back in at 8am PST, and stayed on through the end of the game at 5pm PST, for a total of 16 hours. The bulk of his contributions came during that first 7-hour shift. By the time he logged in for the second shift the team had patches and exploits working on all four services, so it was mostly a passive monitoring of the services and restarting some of them when they went offline, as well as performing backups of our exploit scripts.

Jonathan Chang

Jonathan found the vulnerability for the service Backup (1) at the start of the competition. In response, he wrote an attack script within the first 30 minutes of the competition. He also wrote the attack script for the service Say What (2) after Michael found the vulnerability from one of his logs. He cleaned up the code to Mehran's exploit for Flaskids (3) and updated it to run on every tick, and added functionality at the end that deletes other teams' databases if they haven't changed their passwords from "changeme" (which causes the script bot to not be able to find their flags). He also wrote the final version of the exploit for the service Sampleak (4), in coordination with Michael and Jonathan Ong, after the core parts of the exploit were discovered in a log, which Jonathan Chang worked on with Michael. Jonathan Ong helped in developing the pwntools script.

Jonathan Chang also wrote the initial patch for Backup to sanitize inputs by replacing '*' with a 'q', and then by preventing non-alphanumeric characters (ultimately, Jonathan Ong's encryption patch was more comprehensive). He also changed the password to Flaskids to prevent availability attacks, although that change was only finalized when Josh restarted the service. He participated in the competition for more than 20 hours from the beginning to about 9am the next day, and then got back on in the afternoon from 1pm.

Michael Kotovsky

Michael studied services Say What (2) and Flaskids (3) with Josh and ended up patching Flaskids by parameterizing the SQL statement to prevent injection. Later, he also patched Say What by using the basename() function to prevent directory traversal. He also stayed up to study service Sampleak (4) with Jonathan Chang in the early morning, and eventually reverse engineered team Young-Grasshopper's exploit by looking through the network capture logs. Along with Jonathan Ong and Jonathan Chang, he helped implement the exploit. Michael also patched Sampleak from that vulnerability by changing corresponding bytes in the bytecode. He taught the team, especially Jonathan Chang, how to use screens to keep services online and transferrable to other members even when someone logs off. In addition, Michael coordinated with Josh to maintain Docker container state and service availability. He helped facilitate pass down of information in the very early morning, and also hosted a number of Zoom screen shares to keep members on the same page. He participated in the competition for 14 hours, and was instrumental in providing continuity between the members that logged on Friday and the ones on Saturday.

Jonathan Ong

Jonathan got on early Saturday morning and worked with Michael to get up to date on developments. Around that time, the last service Sampleak (4) finally got exploited, so he and Michael, and Jonathan Chang, rushed to develop a response. Michael found the exploit intact in one of his logs and reproduced the bytecode, and then the three worked in tandem to implement it. Jonathan Ong's suggestions initiated our first version of the code that was able to reach a shell. Later, during a discussion about vulnerabilities for the Backup (1) service, Jonathan revealed that he had completed a patched version of the service that changed the functions to encrypt and decrypt output files. After the team implemented his patch, our logs proved that no one was able to exploit that service further.

Kumar Raj

Kumar initialized his firewall at the start of the competition, then added several keywords such as '*' for the service Backup (1) as an early defense. By monitoring his logs, we were assured of its efficacy. From time to time, he updated firewall rules to protect services, using Michael's logs to analyze other teams' exploits. He also helped in studying the services to pitch ideas to develop exploits.

Mehran Tajbakhsh

Upon Josh's discovery of the service Flaskids' (3) vulnerability, and his subsequent development of the exploit, Mehran expediently implemented the initial version (with all of the main logic in the final version). He also analyzed the disassembly of the service Sampleak (4) and drafted the outline for its exploit. Finding the vulnerabilities in the service Sampleak (4) was one of the most challenging parts of this competition since we only had access to a binary file and no source code was provided.

His well-prepared readme files were also useful in the competition. For example, Joshua faced an encoded PHP file when he wanted to analyze the service Say What (2). Mehran had thought about this scenario beforehand and had a solution outlined in one of his documents, Using that, Joshua managed to successfully decode that file and proceed with his analysis. His efforts helped us have these exploits up and running as early as we could have and gave us an edge in the beginning of the competition.

He actively participated in the PCTF during the first 9 hours, from the setup hour which started at 4pm to 1am PST. Because he worked at his regular job on Saturday, he only got a chance to log back in again at 4pm and stayed with his team during 5pm, for a total of 10 hours.

Summary

Through the efforts detailed above, we were able to achieve 4th place in the competition, a fact that we were all very proud of. We were able to identify short and long term strategies and effectively execute them in the competition. However, we were also able to respond to changing circumstances as they arose, and devised the best way forward for maximum point acquisition. We were proud to report that our firewall, logging and encryption scripts worked as well as we planned, and we reaped the rewards through their use. While the executor, database, checklist and analyzer did not get much use, implementing them allowed us to gain confidence in our overall strategy, so that we knew what we could fall back on if they were required. While we wanted all aspects of the project to be useful, the primary goal of the competition was to win, even if that meant discarding some parts of it on game day. Regardless, the project pushed us to approach the CTF problem from various perspectives, to reconcile with each others' ideas and step outside our comfort zone. It prepared us for the mindset needed in the competition. It made the experience well-rounded and complete.

References

[1] Sanjib Sinha, “Bug Bounty Hunting for Web Security: Find and Exploit Vulnerabilities in Web sites and Applications”, 1st Edition, Apress, 2019

[2] Chris Anley, John Heasman, Felix FX Lindner, Gerardo Richarte, “The Shellcoder's Handbook: Discovering and Exploiting Security Holes”, 2nd Edition, Willey, 2007

[3] Atul Gawande, “The Checklist Manifesto”, Metropolitan Books, 2009