TORCHWOOD 2: Tactically Organized Robotic Code Helping With Offense Or Defense
An Antagonistic Senario Process Isolator

Tom Boning, Eli Davis, Ceres Lee, Steven Valdez

December 13, 2013

1 Introduction

In 6.858, we learned about the dangers of poor coding practices and how they could create vulnerable programs. These are then exploitable by attacks from adversaries that could crash the program or worse, allow an adversary access to user data. In class, we were shown how to craft such attacks and how to modify our zoobar servers to defend against these attacks. However, in our labs, we had direct access to the source code and could modify it directly to defend against antagonistic attacks. For this project, we aimed to create a tool with which we could defend a program that we did not have any prior knowledge about, and are unable or unwilling to modify source code. In this report, summary, whateverwecallthis, we present a system that allows inputs into a vulnerable application to be filtered.

2 Project

TORCHWOOD consists of five main components, four proxies and an overarching framework, that work together to sandbox and secure a process against malicious inputs. At the heart of TORCHWOOD is the protect.py script, which is responsible for intiailizing all the proxies and running them on child threads. The four proxies each monitor a mode of input, creating layers of protection around the application by preventing bad inputs from repeatedly crashing the application

2.1 Protection Scheme

At the heart of TORCHWOOD is a script that reads a configuration file and initializes the various proxies. The configuration file defines what each proxy should be protecting. This information includes which ports should be filtered, what database credentials should be usedused and which files should be watched. Once the proxies are initialized, we collect any arguments to the subprocess call specific to each proxy from the proxies, such as a new socket for stdin. Then, we can fork off a new thread for each proxy to ensure that each proxy can run simultaneously and independently of each other. Finally, we start the protected application with the specified arguments and wait until the protected appliction crashes or exits.

If the application crashes and returns a non-zero exit code, we inform all the proxies and then restart the application. Each proxy updates the arguments to the subprocess and possibly adds new filtering/sanitization rules as a result of the crash. We then proceed to restart the application again. Unfortunately this scheme does not work with poorly written applications that do not exit cleanly on a crash and cannot restart without some external intervention. Additionally, this scheme does not work with applications that exit cleanly when parts of them crash.

2.2 Network Filter

The network filter determines if a network based request is legitimate and then forwards the request to the application's network interface. To do this, we use the iptables rerouting system to route packets to an unused high port number from the port number bound by the application. Then, we listen at the high port number to receive network traffic for filtering, finally sending it back to the application. In order to fiter out bad requests from harming the application, the network filter stores the last network request. If a request triggers an application crash, then that request is added to a list of blocked queries. If the number of requests that it finds is above the threshold, then the future similar requests are blocked. The network filter also saves the list of blocked requests when protection is halted and loads this at startup for the network filter. The network filter also supports multiple input ports and handles each separately.

The current scheme only works for TCP based traffic, but this could be extended to UDP or other proprietary network schemes in a similar manner. Also, the blocking scheme is strictly matching by default, but could in the future extend to fuzzier systems, depending on the application being filtered. HTTP in particular is ripe for better filtering, as it has a defined pulbic protocol that is widely used.

2.3 File Protection

In order to protect data and configuration files being used by the protected application, we have a system that uses a combination of inotify and the Linux Auditing System to watch file access. We then can revert any changes that are not approved. The inotify system sets off notifications whenever a filesystem object is accessed on the machine and lets any listening programs know what sort of operation was performed on the object from an open or a data modification. Meanwhile, the Linux Auditing System, accessible through auditctl, lets us create a log on an object. This logs which program accesses a file and what the program does to that file. We then lookup a file's access log whenever we goet an inotify event to verify that the access was a result of the protected process or if it was modified due to some external factor.

Once we determined whether a file access was legitimate or not, we can then create a backup of the file, assuming the access was legitimate, or restore a previous backup that had been taken. The restoration is only necessary if the illegitimate file access modified the contents of the file. In an extension of this system, we could also remove critical data from being publicly accessible, unless the accessing program is the one we are protecting. However this would require hooking into the system calls that are being made in order to swap out the requested data before the first read by the program.

This system gives us the benefit of preventing random modifications to the protected file's critical data while still allowing the program to run without having to be modified to account for the changing file data.

2.4 MySQL Proxying

For filtering database access, we used a python program which launches the mysql proxy with a custom admin script. The script had several functions. First, it logs all queries to the database in a file that is stored outside of the database in case the database is compromised and deleted. If the python program is informed that the protected program has crashed, it notes the last query to the database. The proxy blacklists any query that was the last query before a crash twice. Blacklisted queries will not be forwarded to the database.

Unfortunately, this approach doesn't work very well in practice; most malicious requests to the database won't cause the program to crash. Thus, in addition to the crash-induced blacklist, the proxy has a series of by-default-blacklisted commands: Anything with a comment is banned as malicious; the legitimate sql querries with comments in them are few and far between. Similarly, any query with an AND or an OR with a statement which evaluates as True will be blocked. We considered blocking more functions out of hand, but those are the only two commands of which we are aware which are only ever used in malicious queries. Ideally, we would sanitize user generated inputs (no name = ' '; Drop tables, for example). Unfortunately, by the time that the proxy gets hold of the query, there is no way to tell which parts of the query were user generated, so this kind of sanitization is impossible.

2.5 Standard Input Filtering

To monitor inputs from other local applications on the server, the standard input filter receives input and checks them before passing them on to the protected application. This is done by using using socketpairs. Socketpairs creates a pair of sockets, so we essentially have a pipe where input fed into one end comes out the other. One end of the pair of pipes is passed to the client process as its stdin. Then, once the standard input proxy has determined whether the input is safe to be passed on, the proxy writes the data it received to the other end of the socketpair.

Initially, we assume all inputs are valid since we cannot make any assumptions about the application we are protecting. As the application runs and crashes, the standard input proxy begins building filtering rules for what inputs are allowed to be passed on. Currently, we ban any inputs that have crashed the application twice. We only ban inputs once they have crashed the appliction twice to prevent accidentally banning valid inputs that happened to be inputed at the same time some other factors, such as network communications, caused the application to crash. Additionally, we define any input that is greater than 256 bytes in length as a long input. If a long input causes the appliction to crash, we start placing length limits on our accepted inputs. The proxy keeps track of how many long inputs crashed the appliction. Each time a long input crashes the application, the proxy uses a weighted average of the current length limit and the length of the most recent input to crash the appliction to calculate a new maximum length.

3 Test Cases

We wrote a few test cases over the course of the project to evaluate the correctness of TORCHWOOD:

3.1 Omega

```
./tests/alpha.py PORT | ./torchwood ./tests/omega.conf
```

First we have the alpha.py program, which simulates STDIN by reading in from a port and printing that out to its stdout. This allows us to test stdin for torchwood without having to restart the whole system for new input.

Omega works as a simple test of all four portions of TORCHWOOD, by repeatedly doing the following:

1. Receive a "password" from stdin and store it in a "32 character" buffer. (Artifically erroring out if the input is too long, due to the difficulty in getting overflow issues in python)
2. Load a configuration file with network details and a 'flag'.
3. Select the user with that password from the MySQL database, not sanitizing the inputs.
4. Print out the selected user(s) to stdout.
5. Start up a "web server" that looks for requests of the form "METHOD PATH VERSION" from the client and returns a fake webpage with the flag. If if the PATH has "quit" in it, restart the process.

This provides a full test of all of the relevant portions of TORCHWOOD, with long and invalid input to stdin resulting in an exception, invalid configuration files resulting in a crash, unsanitized MySQL requests resulting in information leakage, and badly formatted requests to the "web server" resulting in parser crashes. While applications like this are unlikely to be found in the wild, this provides a good display of the impact of TORCHWOOD on reducing crashes and vulnerable attack surfaces.

3.2 Basic HTTP Server

```
./torchwood ./tests/basic_http_server.conf
```

This test is a basic python HTTP server that responds to both GET and POST requests from a client. Additionally, on a GET request of /CRASH it closes it's socket, crashing the server due to socket errors. This application is successfully protected by TORCHWOOD, using the strict request blocking implemented by default.

3.3 Zoobar 6.858 Application

Unfortunately, the Zoobar application does not work with the TORCHWOOD proctection scheme due to zookd and zookld's handling of crashes from spawned threads. This prevents the program from crashing on malicious input, such as a buffer overflow caused by a long request path. Since this information is not transmitted outside the Zoobar application, TORCHWOOD does not detect the crash by default.

4 Future Work

In the future, TORCHWOOD could be expanded to include cleverer heuristics for figuring out what inputs are bad and blocking them or flagging them as potentially dangerous. In addition, some work can be done in detecting crashes and the type of crashes that a program undergoes, to provide better information to the proxies, allowing them to adapt to crashes in a more intelligent way. Another extension is to use the blocked bad requests in novel ways, such as extracting the exploit from the blocked requests and using to for offense during Cyber CTF competitions.