Title

By Dirk Cummings

Orion Miller

Senior Project

Dr. Philip Nico

December 2, 2011

Abstract

Abstract

Contents

1 Research of other CTFs

2 Objectives

3 The Duke Nukem Scenario

3.1 Objectives

3.2 Design

3.3 Game Day Results

3.4 Problems and Solutions

4 The Star Trek Scenario

4.1 Objectives

After the first CTF competition, we wanted to make the next CTF focus mainly on solving two problems:

- testing the individual's coding and hacking skills rather than their
- ability to find and use pre-built tools and prevent the participants from breaking the servers (rather than the services) and ruining the game.

These would not only improve the overall game play and enjoyment, but would help eliminate game town time spent repairing and restoring the virtual machine server's images. Additionally, by designing new services from the ground up we can demonstrate to participants and give them experience with the common cause of vulnerabilities: poor design.

The first scenario was good for getting the uninitiated starting in security, however, it did lacked While the first scenario focused mainly on attacking servers and their services, we needed to even the importance of defending to better align the game with a real world scenario.

Lastly, players of the first competition noted in their feedback they wanted more teamwork based cooperation. Specifically, those with little to no security experience had a tough time working out the solutions and would have liked to work with others outside of their team to learn techniques and practices. Team formation in the second competition needed to move away from the standard method by which those who know each other tightly group together thus separating those who know from those who don't. This method tends to create groups with varying experience and a hording of knowledge.

4.2 Design

To ensure no participant would be able to use pre-built tools to exploit vulnerabilities, we chose to design services one might program during the first implementations of a ship from Star Trek. This design was chosen for two reasons: in the summer of 2011 Netflix began streaming every episode of every Star Trek series [1], and we couldn't find anyone who had created such services or written anything to exploit such services. A listing of these services and their designs can be found in the

Game Play and Services sections. The new approach also allowed us to make changes to the game play to focus on a more team based play.

4.2.1 Game Play

The new game design was made to resemble a WarGame style of play consisting of only two teams each their own server to manage. We were even fortunate enough to provide both teams with their own rooms to work in together. Each team would initially receive the same source code of the services to fix and deploy on their servers (each a virtual machine running Ubuntu Server 11.10). Their goal to find and patch the vulnerabilities in their code and exploit them in the other team. If a team could keep their services responding and pass basic functionality tests they would score points. However, if their services become unresponsive or failed the tests, whether their own fault or the opposing team, they would stop receiving points while the opposing team would score additional points.

We also wanted to give those participants with little or no hacking, or even coding, experience something entertaining and contributory to do. Since these services should be fully functional and represent a star ship, then it stands those who feel they can't contribute could fly the ship around and score points for their team by shooting the other team's ship.

4.2.2 Score Keeping

4.2.3 Services

On competition day, there were three implemented services in production for teams to work on with three additional services planned to be released later in the competition. Each of the planned services and a brief description of their functionality is listed at the end of this section.

Services are designed with two major parts, the built-in (or base) functionality, and the competitor's implementation.

The base functionality is implemented by us and provided in unlinked pre-compiled form for each team build from. It is meant to represent the initial implementation of ship's services programmers. There are a few seeded vulnerabilities and even more poor and hastily made designs. Base implementation of a service is responsible only for:

- provide only the most basic level of support and implementation necessary for the service to run,
- setup the network support required for running the service,
- listen on an assigned port, accept incoming connections, and pass on the connection to a handler which processes the incoming data,
- run base functionality tests on the competitor's implementation and pass on the results to the Score Keeper for scoring,
- and keep a modest amount of internal book keeping specific to the service to ensure the honesty of a competitor's implementation.

While the exact functions for each service differ, every service's base implementation has functions (defined externally in header files) for the competitor's code to call before finishing processing an incoming request. For each of a service's base function, there is a corresponding competitor's wrapper function to be called before the equivalent base function. The location of the wrapper functions are stored in a structure as function pointers defined in a shared header file between both implementations. These wrapper functions allow the competitors the chance to perform their own book keeping and sanitize inputs from requests before calling the base functions. A diagram of the typical call stack is available in Figure 1. This is one way in which the players can patch vulnerabilities and design flaws.

To communicate between and control other services, each service had a pre-defined packet structure mimicking an IP packet. Both Teams started off with the same structure but were allowed to add on to fulfill any future needs. The initial structure contained only enough fields to control each specific command of a service. Any security or checksum features were left up to the participants to develop and deploy.

4.2.3.1 Power The critical service of each ship responsible for managing and distributing a fixed amount of available ship power to all other running services. A service may only run while it has been allocated power. Should a service lose power, it shall be stopped and non-responsive until power is restored. If the power service itself is shutdown or crashed, all other services would be taken offline.

4.2.3.2 Engines Just like the engines in the star ships with support for both impulse and warp speeds. The engine service is only concerned with managing either engine's current power allocation and speed as well as ensuring their health. Each engine leaks varying amounts of radiation which if not dissipated could damage the engines to the point they are no longer functional. Scotty can fix engine damage, but he's not a bloody miracle worker; he can only work so fast.

4.2.3.3 Navigation Responsible for setting a ship's course direction, managing the ship's internal representation of the map, and managing the engines.

4.2.3.4 Communications The communications service is independent of all other services except for the power service, and instead of managing some aspect of a ship, presents a team with a number of cryptography puzzles to be decrypted, solved, and posted back to the Score Keeper for validation.

4.2.3.5 Weapons

4.2.3.6 Shields

4.3 Game Day Results

The competition started around noon on November 5th, 2011 with four participants and three of the services in production. These were the power, engines, and navigation services. The others had not yet been implemented, but were planned to be release some time at the end of the first day or during the start of the second day of competition.

Over the next three or four hours both teams were still reading through the documentation and asking questions trying to figure out the structure of the game and the code. Very little code was actually being written, but design flaws and vulnerabilities were slowly being realized. One or two members of Team A had previous experience in UNIX System Administration and immediately began setting up monitoring tools to track incoming requests and log the commands executed by each service. After which they moved their focus to the operating system running their services and attempted to "secure" it instead of their services. A few hours later and they abandoned this approach.

Team B on the other hand, situated in the neighboring conference room decided to start by developing tools to help them manage the services. (The initial implementation provided no user friendly means to control a service. A member would have to format a packet, open a connection to the service, send the byte stream through the connection, and process any returned information.) Through this management tool the team was able to track command requests between services and drop unexpected incoming requests. Given the sequence of events for each service's commands, the team could select the authentic commands and dump any malicious ones. This was a great start for the team as they were able to reap quick results.

Up until this point Team A had been setting up connections and sending random bytes of data to Team B's services. Since the original implementation was faulty, these connections and incorrect data often crashed Team B's services, causing them to fail tests and costing them points. When the new filter went into place, these malicious attacks were almost completely eliminated.

Later on that night Team B started on their design for a basic level of encryption.

4.4 Problems and Solutions

Lack of Experience Among Players No one here really knows how to exploit code, and those that do didn't participate.

For the first event we offered Security 101 sessions every Saturday leading up to the competition to help prepare players and get them inspired. Because of our lack of availability we weren't able to get a similar series setup. We don't want to suggest correlation implies causation, but there was an obviously lower attendance for the second competition.

Stagnation While one team outright dumped all incoming requests and the other did essentially the same thing by hardening access to their services no one not ssh'd into their server could gain control. Ultimately a good achievement, however, game play began to stagnate.

An easy fix would be to require all services to have an open public interface. Instead of giving teams control of the server and the ability to lock others out, anyone should be able to control a Team's services. This does bring up the possibility of each team's ship experiencing erratic behavior (such as rapid navigation changes and little to no movement across the game's map) but this can be redefined as desired behavior. Such behavior can serve as motivation for a team to work on ways to prevent the attacks or increase the attacks on another team to bring their ship down first.

Short On Time One quarter to design, implement, and test completely custom services and wrap them up into a game is not enough time for only two people. You say you're going to work on it over the break, but good luck keeping priorities.

At least two more people and one more quarter would help prevent slipping and bad code (see next problem).

Bad Tests We started performing our implementation tests one and a half weeks prior to the competition. While were able to catch and solve a few big problems before the last minute, we weren't able to perform a "dry-run" of game and iron out some other major issues between the Score Keeper and the services. A fare and functional Score Keeper is key to a successful CTF []. Without the dry-run, we weren't able to find logic and design flaws in our service tests and reporting which severely affected game play.

Ideally, all code (services, Score Keeper, helpers, etc.) should begin undergoing final implementation tests and debugging four weeks out from the competition and integration tests done two full weeks out culminating in a complete dry-run of the game. All services and Score Keeping should be fully implemented and tested before integration testing begins. Additionally, at least one virtual machine image should created and saved for each of the following (and will not be used for the competition):

- a clean (without any game code) and fully configured server
- for each team, an working (with all game code included) instance which also contains and passes all implementation and integration tests
- for each team, the final working copy distributed at the start of competition

Code Management Each team's code had slightly different configurations which had to be changed by hand within a single code base. With 10 to 12 changes to make in each service's code, insuring all changes were properly made before each update was difficult under pressure. The design of our Score Keeper (see Section 4.2.2) required all service specific configurations to be contained within the pre-compiled source code provided to each time. On a few occasions configuration settings for one team made it into the compiled version of the other team. Resulting in services not starting up, not communicating properly with other services, or communicating incorrect results to the score keeper.

Ideally the services would have been written such that no updates were ever needed to be pushed to servers during game play, however, other problems inhibited this solution.

Because of the way we decided to authenticate service status updates with the proper team (as described in Section 4.2.2), we would not be able to keep all configurations in a common header file. Instead we should have either created two code bases (one for each team) with configurations which don't change, or use a simple script to write all unique configurations, compile the service, and push the update to the team's server. The former would introduce the problem of ensuring both code bases were synchronized with the proper updates. A simple solution, but one just as error prone and no less irritating. The latter solution would take little time to implement and cut update hassles down to a simple command line execution.

Open-BSD During the first competition, we had a number of people trying to attack the server's services rather than finding the vulnerabilities we created. Servers in the first competition were unpatched Ubuntu Server 10.10 instances with a number of open network services. Since this go around we didn't need things like PHP, Apache, MySQL, etc., a stripped down operating system capable of compiling code and running our network processes was ideal. After the initial configuration, users would not have sudo-er privileges or be able to install new packages.

However, during our integration testing, we ran into a number of problems getting the services to start up and behave properly. All code was POSIX compliant but services had difficulties opening and binding sockets as well as sending and receiving data. Limited remaining time (see **Short On Time** and **Bad Tests**), we were not able to solve the problem and instead switched to fully patched Ubuntu Server 11.10 instances with no sudo-er privileges and without Apache, PHP, or MySQL installed.

5 Comparing Scenarios

6 The Next Ideal Solution

6.1 Game Play

6.2 Network

6.3 Services

6.4 Score Keeper

6.5 Teaching and Timing

Security 101 sessions the quarter of the competition.

Advertise every week (in conjunction with the 101 sessions).

Reserve Bonderson rooms WELL in advance and confirm the reservation months in advance.

The end of week 6 in a quarter might be the ideal time to avoid conflicts with midterms. However, use best fit (least number of conflicts) when picking a date avoiding sporting events, club functions, holidays, midterms, etc., while knowing no weekend you chose is ever going to be perfect.

No all nighters. You're mainly going to be thrashing and you'd rather be up all night during the competition rather than working on only a few hours of sleep and heading towards 56 hours without sleep.

7 Conclusion

8 Bibliography

References

[1] A. Pascale, “Netflix begins streaming star trek tos, tng, voyager and enterprise,” July 2011. [Online]. Available: http://trekmovie.com/2011/07/01/netflix-begins-streaming-star-trek-tos-tng-voyager-enterprise/

```
extern struct PowerFuncs {

   void *(*request_handler) (void *confd);

      The following functions are your wrappers around the built-in / main
        Power service functions. They will be used for testing your service
         implementation and will not be called until your write your own
         implementation. At the end of each of your implementations, you must
         call the external equivalent functions. A diagram of the setup is
         bellow along with an example of the order of the function calls.

      Diagram:

      ----------------------------------------------------------
      |  request_handler                                       |
      |                                                        |
      |     ---------------------------------------------------|
      |     |  additional_function                             |
      |     |     (if you want to add additional               |
      |     |     checking or functionality to add security)   |
      |     |                                                  |
      |     |     ---------------------------------------------|
      |     |     |  wrapper_function                          |
      |     |     |     (for your book-keeping)                |
      |     |     |     EX) pow_funcs.wadd_power(int, int)     |
      |     |     |     ---------------------------------------|
      |     |     |     |  built-in_function                   |
      |     |     |     |     EX) add_power(int, int)          |
      |     |     |     |                                      |
      ----------------------------------------------------------

      Function Call Example:

      --> Incoming Connection Request
            |--> Connection Accepted
            |--> pow_funcs.request_handler(connection_file_descriptor)
               |--> additional_function(...)
                  |--> pow_funcs.wadd_power(int, int)
                     |--> add_power(int, int)
                  |--> send_packet(struct PowerHeader, socket_fd)

   int (*wregister_service) (int, int);
   int (*wunregister_service) (int);
   int (*wadd_power) (int, int);
} pow_funcs;
```

Figure 1: A 0 of the provided header file for the Power service. It describes the structure and use of the wrapper functions as well as demonstrating the data flow of incoming requests.