Charger Active Defense v1.0 Team 2 - Group 12

Noah Sickels, Adam Brannon, William Lochte

## Background / Abstract

Modern attack tools are highly efficient, while the cybersecurity industry struggles with active defense capabilities. Current passive defenses primarily focus on mitigating threats through means that can disrupt business operations or can be circumvented. Additionally, with the rapid advancement of artificial intelligence and prompt engineering, attackers can instantly generate various attack tools to target organizations.

The Charger Active Defense project focuses on developing a network-based fuzzing workflow to effectively and comprehensively test these known and AI/LLM-generated attack tools. It aims to identify responses that may cause the attacking application to crash or hang. Responses generated could be saved and sent back to the adversary through a Python replay service during a detected attack. Due to the multi-threading nature of many attack tools like Masscan and Medusa, effectively fuzzing these tools is complicated. If a fuzzed response leads to a crash or hang, it typically occurs in a thread separate from the attacking application's instance. This crashed thread may either be reported as a false positive or not counted at all. Because of this discovery, at the customer's request, we are pivoting towards thoroughly fuzzing generated attack tools from large language models.

Since we have successfully applied the fuzzing workflow to Masscan, we will also pursue the integration of ThreadSanitizer (TSan) to determine if current fuzz testing is properly crashing threads. The end goal of this workflow is to assess how we can broaden this method to apply to other attack tools, providing organizations with a strategy to defend their systems against an ever-evolving threat landscape.

## Current Project Status, Issues, & Short-Term Activities & Goals

This reporting period consisted of researching and integrating ThreadSanitizer into our existing fuzzing workflow, using the updated fuzzing workflow to start AFLnet fuzz testing on Masscan, and developing the user interface and logging modules for the Python service. At the request of the sponsor, previous AFLnet fuzz testing with Masscan was stopped due to inconclusive results. After modifying the Masscan Makefile, we added ThreadSanitizer into the compilation and instrumentation process with AFLnet and we started the full fuzz testing again.

Our short-term goals for this period were:

* Integrate ThreadSanitizer into Masscan and existing fuzzing workflow.
* Write the user interface for the Python service application.
* Write the error and traffic logging module for the Python service application.

During this period, we encountered a few issues:

* In order to instrument Masscan with AFLnet, we have to use the custom afl-gcc compiler – a wrapper for the existing GCC tool. While we were able to instrument Masscan without issues as-is, when we added the ThreadSanitizer compiler flags, AFLnet instrumented it correctly, but the ThreadSanitizer output did not appear at all. We believed this was either due to AFLnet overwriting all terminal stdout and stderr to maintain the fuzzing test summary table, or that ThreadSanitizer was not redirecting any stdout or stderr to the terminal at all. We were able to confirm that the ThreadSanitizer flags were present in the afl-gcc compilation output and were eventually able to redirect any output from ThreadSanitizer to a separate file by using the `&>` redirect flag in the command used.
* The library we chose for the user interface is incredibly robust, but there were certain features we wanted to add that were problematic. Namely, the status bar/tool bar at the bottom of the terminal window. This status bar displays the current connection status with the target, the network adapter traffic in use, and the port number of traffic being monitored. We eventually were able to get everything to work to redraw the tool bar appropriately for each command (`set interface`, `set ip`, `set port`, `stat/stop`, etc.).

For the next reporting period, our short-term goals are:

|  |  |
| --- | --- |
| Responsibility | Task |
| Noah Sickels | Develop the Python service response handling module, including connectors for desired database and logger module. |
| Adam Brannon | Develop the 1st attack tool (port scanner/banner-grabber) using the Phind model. |
| William Lochte | Develop the 1st attack tool (port scanner/banner-grabber) using the Copilot model. |

## Milestone Status Summary

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Milestone / Task | Projected Due Date | Completion Date | Status | Notes |
| Integration of Masscan with ThreadSanitizer into Fuzzing Workflow | 1/31 | 1/31 | Complete | ThreadSanitizer was added to the Masscan Makefile, and it is now part of our automated fuzzing workflow script. |
| Develop Python Service User Interface | 2/3 | 2/3 | Complete | The core of the user interface is complete but may be adjusted slightly as we continue to develop the Python service. |
| Develop Python Service Error & Logging Module | 2/7 | 2/7 | Complete | Logger module works correctly and will be used throughout the program going forward. |

## Level of Effort / Individual Responsibility Record

|  |  |  |
| --- | --- | --- |
| Member | Hours / Period | Total Hours / Spring |
| Noah Sickels | 28 | 43 |
| Adam Brannon | 18 | 23 |
| William Lochte | 20 | 27 |
| Total | 66 | 93 |

|  |  |
| --- | --- |
| Member | Individual Accomplishments |
| Noah Sickels | Researched implementation of ThreadSanitizer into GCC programs.  Developed the Python service user-interface and command methods.  Added updated Makefile for Masscan to the fuzzing workflow script.   Began updated fuzz testing of Masscan with AFLnet and ThreadSanitizer.  Helped with design and formatting of project poster. |
| Adam Brannon | Researched implementation of ThreadSanitizer into GCC programs.  Developed the Python service logging module for errors and future traffic monitoring.  Helped with design and formatting of project poster. |
| William Lochte | Researched implementation of ThreadSanitizer into GCC programs.   Written docstrings for each of the methods used for the user-interface and the logging modules.  Helped with design and formatting of project poster. |

## Milestone Completion & Analysis

* Updated fuzz testing with AFLnet and Masscan was started with the ThreadSanitizer integration. This included updating the existing Makefile within Masscan as shown in Figure 1 below. A snippet of the ThreadSanitizer output from our initial testing is also provided in Figure 3 below.

A screen shot of a computer code

AI-generated content may be incorrect.

Figure 1: Snippet of Modified Masscan Makefile

* When instrumenting with AFLnet, we were able to confirm that the flags were correctly used from the Masscan Makefile.

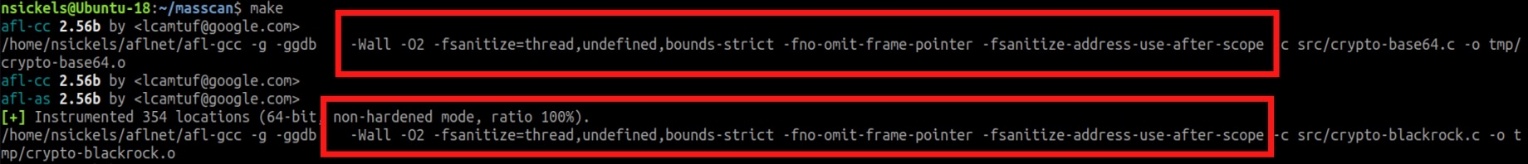


Figure 2: Afl-gcc Compiler with ThreadSanitizer Flags

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 3: Snippet of ThreadSanitizer Output on Masscan

* The Python service user interface is complete, but we may decide to make slight changes and add new commands as the project progresses. Figure 4 below shows the banner and welcome message printed at the start of the program, the user shell prompt, the command suggestion and auto highlighting, and the status bar/tool bar that shows connection status (connected or disconnected), the current IP address of the host, the network adapter actively being monitored, and the port of the service being monitored.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 4: Python Service User Interface

* The logger module is a custom wrapper of the existing Python logger library that adds colorized message types, time and date information, the current logger in use, and the provided text. This implementation includes three additional message types, “SENT”, “RECEIVE,” and “RESPONSE.” These message types correspond to sent traffic, received network traffic, and any sent Active Defense Response. These are specifically intended for later use with the full network handling integration planned later in the project. Example messages are shown in Figure 5 below. Note that there is a distinction between the WARN and RESPONSE message colors – RESPONSE blinks in 1 second intervals but could not be adequately shown here as an image.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 5: Python Service Logger Module

A screen shot of a computer program

AI-generated content may be incorrect.

Figure 6: Custom Message Types Code Snippet

A screen shot of a computer program

AI-generated content may be incorrect.

Figure 7: Code Snippet of Custom Color Wrapper for Logging Library

### Mitigations & Contingency Plans

* N/A – No missed milestones for this period.