# Automatic Program Analysis

CS390R - UMass Amherst

# Course Information

- Project 4 due April 18
- Gradescope Presentation Checkin Sunday

# Today’s Content

- Fuzzer theory
- Setting up a fuzzer and using it
  - AFL++ (Project 5 README explains it as well)

# Basics

- Origin
  - 1988 class project at University of Wisconsin
  - Crashed ~30% of tested unix utilities, and triaged the crashes
  - Paper published in 1990
- This fuzzer was very dumb, completely random
- Provide invalid/unexpected random data to program to find exceptions
- AFL
  - Released in 2014, first public coverage guided fuzzer
  - Very easy to use
  - Brought fuzzing field back alive bigtime

# Types of fuzzers

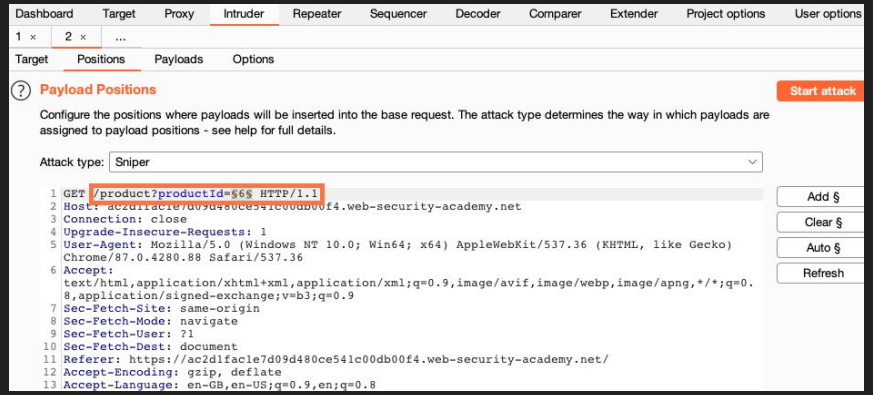
- Blackbox Fuzzing
  - Blindly send input into a binary and hope something crashes
  - Simplest form of fuzzing, but still surprisingly effective
- Greybox Fuzzing
  - Apply various instrumentation techniques to the target to improve fuzzer
  - This makes the fuzzer much more effective in many circumstances
- Whitebox Fuzzing
  - Whitebox fuzzing relies on having access to source code of the target
  - Might use symbolic execution to basically transform target program into a mathematical equation and attempt to solve it
  - Much slower than previously mentioned approaches

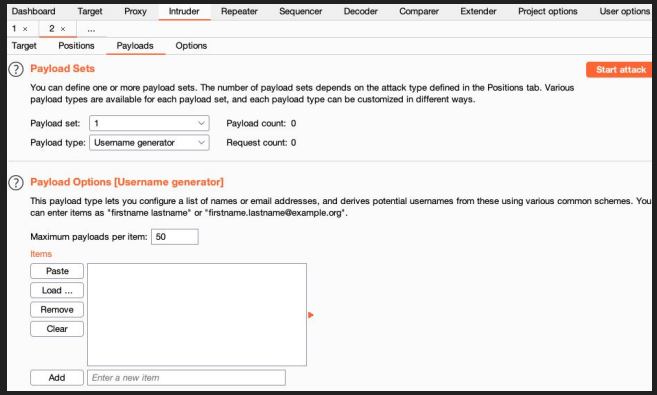
# Mutational vs Grammar-based Fuzzer

- Mutational
  - Very useful to fuzz applications that take input files and parse them to work off of the data
  - Works off of existing “corpus”
  - Randomly mutates the files in the corpus, and runs the target with the mutated file as input
  - Target crashes -> Save input file for manual inspection
- Grammar-based
  - This technique is necessary when attempting to fuzz applications that have very strict rules for their input
  - Compilers, Browsers
  - Does not use a corpus, instead generates inputs from scratch using grammar rules
  - Target crashes -> Save input file for manual inspection

# Web Fuzzing

- Can be used to find vulnerabilities like xss & sql injections
- Often done with word lists containing various sqli and xss payloads that are likely to find bugs
- Some examples:
  - Gobuster/Dirbuster - Bruteforce directories/urls (eg. finding hidden www.some_site/admin.php)
  - Burp Intruder - More control to fuzz specific fields with varying options



# Address Sanitization

- With the fuzzing approaches we have discussed so far, we are only able to detect crashes
- Many bugs such as slight out of bounds reads/writes do not actually result in crashes, they can however still be very relevant security concerns
- With address sanitization, even 1 byte out of bound reads/writes can be recognized and reported
- This is usually done either through compile time instrumentation or emulation

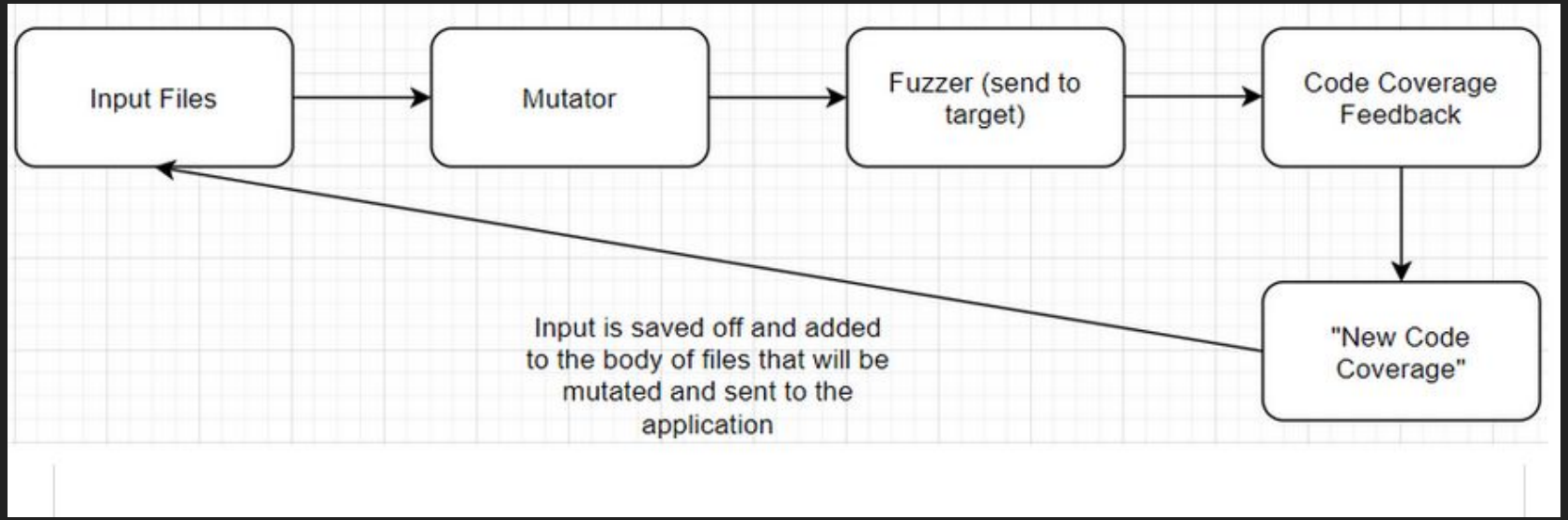
```c
int func(char *x) {
    if (x[0] == '\x41') {
        if (x[1] == '\x42') {
            if (x[2] == '\x43') {
                if (x[3] == '\x44') {
                    printf("Success\n");
                    return 1;
                }
            }
        }
    }
    return 0;
}
```

# Coverage Tracking

- Measure which parts of the program are being executed
- Collection methods
  - Compile-time instrumentation
  - Intel PIN / Intel PT
  - Randomly request fuzzer location at time intervals
- Types of coverage:
  - Block
  - Edge
  - N-gram Edge
  - Path
  - Data
- Why do we care about coverage? (example)

# Coverage Guided Fuzzing

- Different heuristics used by modern fuzzers
- Main idea is to add “interesting” inputs to the corpus, so future fuzz cases make use of these cases



# Seed Scheduling

Often heuristic based depending on a couple different metrics

- Static
  - Graph centrality analysis - assign weight based on the number of reachable edges from a given seed
  - Determine weight based on if seed is on the path to a frequently vulnerable function (eg. memcpy)
- Dynamic (Power Schedules)
  - Assign weights based on input properties (execution time, shorter, more frequent coverage increases, etc)
  - Use mutation history to decide when to stop focusing on “hard’ edges

# Corpus Management

- Corpus minimization
  - Delete “slow” entries from corpus
  - Pros: “faster” corpus
  - Cons: potentially less state
- Initial seed selection importance
  - Having a good initial corpus greatly affects the performance of the fuzzer
- Finding seed files:
  - https://datacommons.anu.edu.au/DataCommons/rest/records/anudc:6106/data/
  - Writing a web crawler

# Mutational Strategies

- Feedback loop approach - use ‘good’ mutations more frequently
- Havoc: Apply multiple randomly selected mutators simultaneously on inputs
- Individual Strategies:
  - Bit flips
  - Byte exchanges
  - Simple arithmetics (+-x on each byte)
  - Known integers
  - Changing size of input
  - Dictionary of interesting strings
  - Splicing - combine two different inputs at random locations

# Crash Triaging

- Crash exploration
  - Separate mode of fuzzer that takes a crashing seed as its input, and attempts to find more crashes based on this input
  - Once multiple crashing inputs are gathered, statistical analysis can be performed to find common cases and to better understand the bug
- Deduping Crashes
  - Group “similar” crashes together to avoid looking at hundreds of similar crashes
- Debugging
  - Simplest: Load into gdb and rootcase
  - Improved: Timeless debugger to step backwards and more easily root-cause the bug

# Harnessing

- Many programs don’t just take a file as input and operate on it
- In these cases, we need to write a wrapper around the target that allows our fuzzer to interact with it
- Examples include gui tools, embedded devices, libraries,

# Performance

- Persistent Mode/Snapshot fuzzing
  - Fuzz in short loop around target functions by saving memory/register state right before this function, to then base future cases off of this location
- In-memory Fuzzing
  - Many targets attempt to read data from disk, loading corpus entirely into memory instead and injecting fuzz cases can greatly improve performance by reducing disk I/O
- Scaling
  - Real world generally runs fuzzers on at least 50-100 cores
  - Can’t use too much shared information between threads (corpus/coverage/statistics)
  - Avoid executing syscalls in fuzz-loop since that can quickly trigger kernel locks