Secure Programming
Security evaluation of software

Leonardo Regano

leonardo.regano@unica.it

PhD Course on Software Security and Protection

University Of Cagliari, Italy

Software vulnerability analysis

- finding vulnerabilities that can be potentially exploitable
- similarly to an attacker, but…
  - … we have the source code
  - … we know our application
- penetration testing: behave like an attacker to test our application security
  - typically separated/external (red) team
  - has limited knowledge of the target application

Software vulnerability analysis: static vs. dynamic

- static analysis
  - analyse the source code / binary
  - no need to run the application
  - (potentially) check program behavior regardless of user inputs
  - example: syntactical checks, formal verification
- dynamic analysis
  - test a running application
  - check program behavior for a finite set of inputs
  - examples: fuzzing, penetration testing

Software vulnerability analysis: white-box vs. black-box

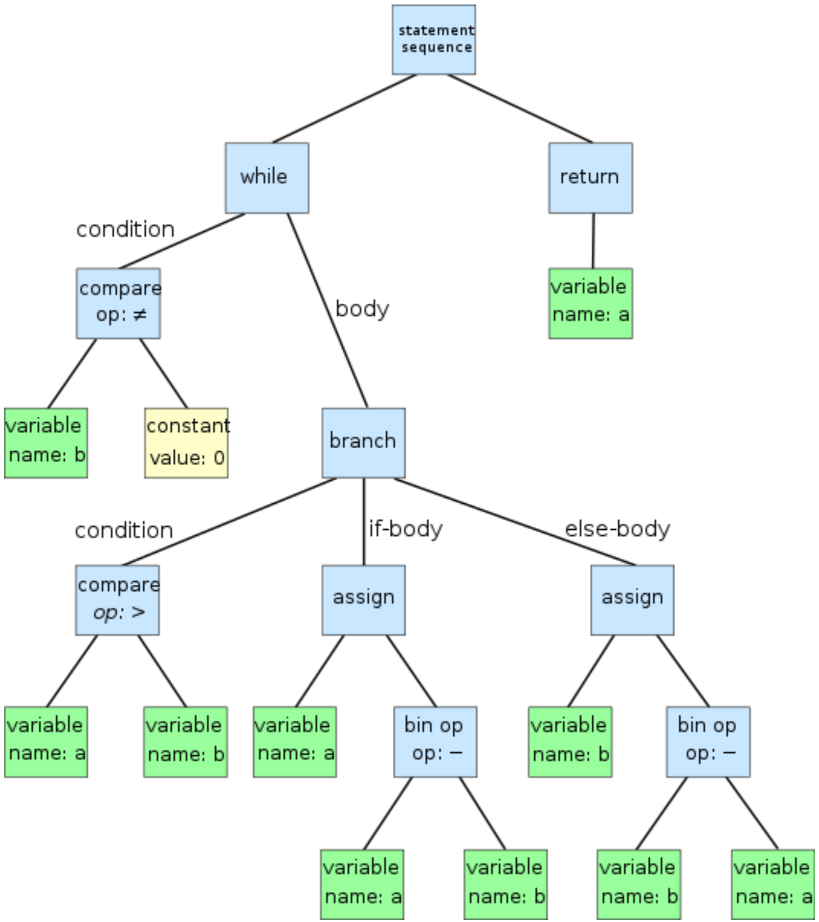
- white-box techniques
  - complete knowledge of the target application
    - source code
    - specifications (requirements, business logic, etc.)
    - design documentation
- black-box techniques
  - no or limited information about the target application
  - similarly to the attacker
  - e.g., penetration testing, analysis of closed-source external libraries
- gray-box techniques
  - a mix of white and black box techniques
  - e.g. greybox fuzzing
    - fuzzing guided with info obtained from source code

Source code models

- three phases
  - lexical analysis
    - result: tokens (i.e. statements)
    - search for vulnerable functions (e.g. strcpy)
  - parsing + semantic analysis
    - result: Abstract Syntax Tree (AST)
    - result: symbol table (variable, constant, functions …)
- parse tree + AST + symbol table enable more checks
  - type checking
  - style checking
- AST + symbol table can be used to build more human-friendly representations
  - e.g. Control Flow Graph, Call Graph
  - simplify human inspection…
  - … for code comprehension and vulnerability hunting

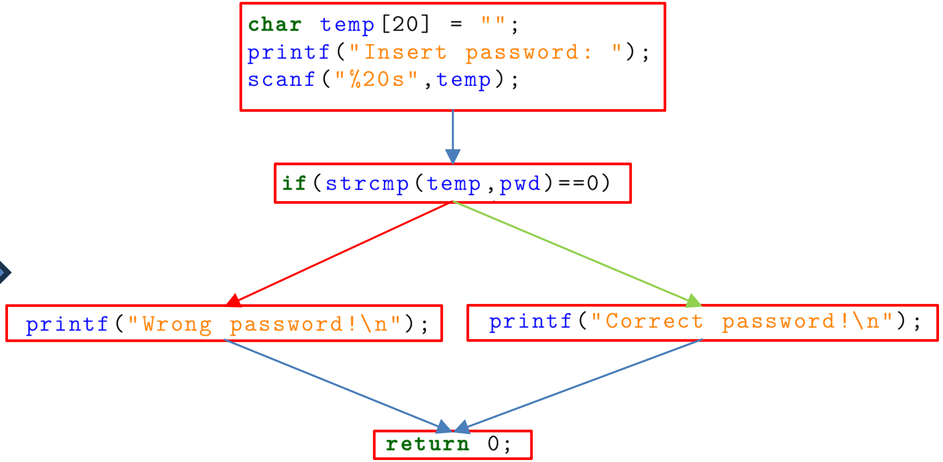
Abstract Syntax Tree example

```
while(b!=0) {
        if(a > b)
                a = a – b;
        else:
                b = b – a;
}
return a;
```



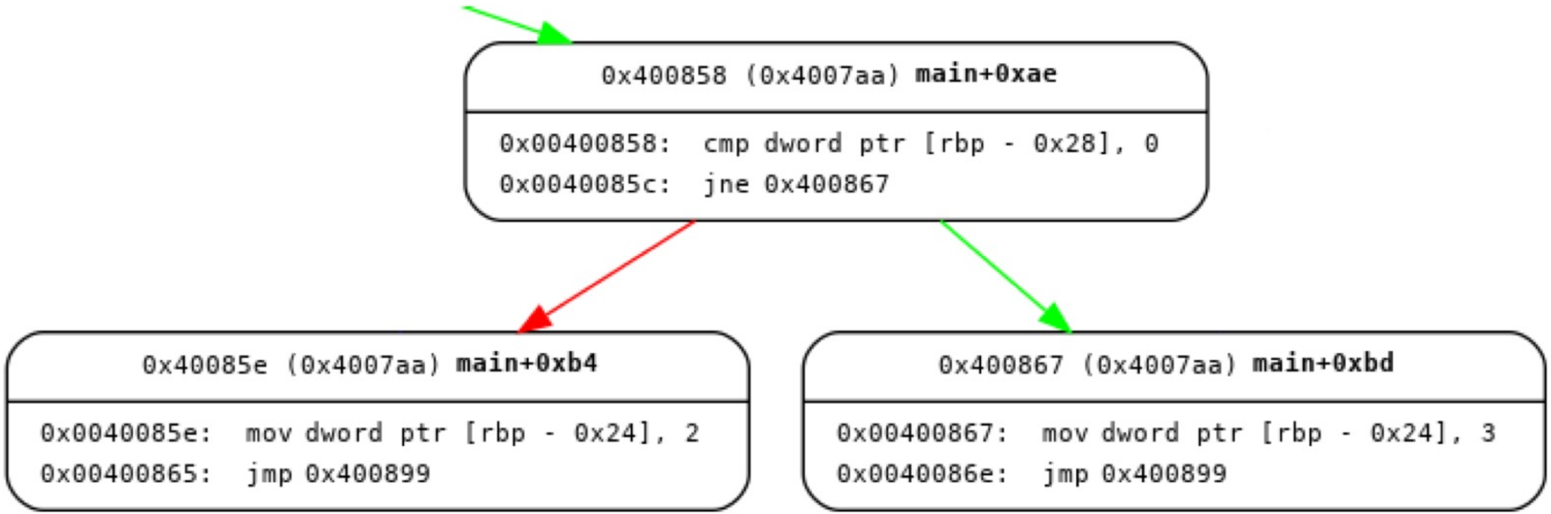
Control Flow Graph from source code

```c
#include <stdio.h>
#include <string.h>

char pwd[] = "hardcodedPassword";

int main()
{
        char temp[20] = "";
        printf("Insert password: ");
        scanf("%20s",temp);

        if(strcmp(temp,pwd)==0)
                printf("Correct password!\n");
        else
                printf("Wrong password!\n");
        return 0;
}
```



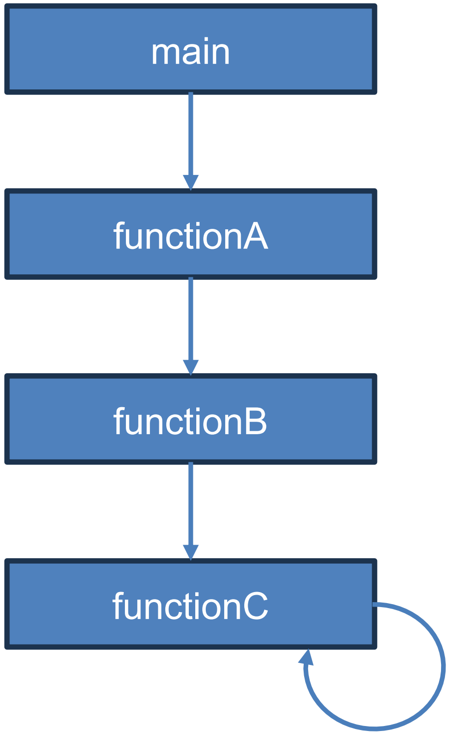
Basic Block (BB): sequences of consecutive instructions, so that if the first instruction of a basic block is executed, the other instructions in the basic block must be executed as well, in the specified order

Control Flow Graph from binary code (x86 ASM)



Call Graph

```c
#include <stdio.h>

int main() {
    functionA();
    return 0;
}

void functionA() {
    printf("Function A called.\n");
    functionB();
}

void functionB() {
    printf("Function B called.\n");
    functionC(3);  // Example with n = 3
}

void functionC(int n) {
    if (n > 0) {
        printf("Function C called with n = %d.\n", n);
        functionC(n - 1);  // Recursive call with n-1
    }
}
```



Taint analysis (or propagation)

- static analysis that can be aided by CFG
- problem: find if a source can propagate to a sink
  - source: input received by a program
  - sink: program statement that, if dependent on user input, causes a vulnerability
- we *taint* a variable
  - meaning that we know it takes it value from user input
- we follow all the possible program flows through the CFG
  - a statement may propagate the taint to another variable
    - e.g. strcpy(new_tainted_variable,source)
  - a statement may clean the taint on a variable
    - if the variable is assigned a new non-tainted value (e.g. strcpy(tainted_variable,”hello”)
    - if the statement sanitizes input
- if we taint the sink, we found a vulnerability!

Taint analysis example

```c
int main() {
    char userInput[100];
    getInput(userInput);
    processInput(userInput);
    return 0;
}
```

```c
void getInput(char *input) {
    printf("Enter a command: ");
    fgets(input, 100, stdin);
    // Remove newline character if present
    input[strcspn(input, "\n")] = '\0';
}
```

```c
void processInput(char *input) {
    char command[200];
    // Unsafe concatenation (vulnerable to command injection)
    snprintf(command, sizeof(command), "echo %s", input);
    executeCommand(command);
}
```

```c
void executeCommand(char *command) {
    printf("Executing command: %s\n", command);
    system(command); // Unsafe use of system() with tainted input
}
```

Taint analysis example

```
int main() {
    char userInput[100];
    getInput(userInput);
    processInput(userInput);
    return 0;
}
```

```
void getInput(char *input) {
    printf("Enter a command: ");
    fgets(input, 100, stdin);
    // Remove newline character if present
    input[strcspn(input, "\n")] = '\0';
}
```

source

```
void processInput(char *input) {
    char command[200];
    // Unsafe concatenation (vulnerable to command injection)
    snprintf(command, sizeof(command), "echo %s", input);
    executeCommand(command);
}
```

```
void executeCommand(char *command) {
    printf("Executing command: %s\n", command);
    system(command); // Unsafe use of system() with tainted input
}
```

Taint analysis example

```
int main() {
    char userInput[100];
    getInput(userInput);
    processInput(userInput);
    return 0;
}
```

```
void getInput(char *input) {
    printf("Enter a command: ");
    fgets(input, 100, stdin);
    // Remove newline character if present
    input[strcspn(input, "\n")] = '\0';
}
```

source

```
void processInput(char *input) {
    char command[200];
    // Unsafe concatenation (vulnerable to command injection)
    snprintf(command, sizeof(command), "echo %s", input);
    executeCommand(command);
}
```

```
void executeCommand(char *command) {
    printf("Executing command: %s\n", command);
    system(command); // Unsafe use of system() with tainted input
}
```

Taint analysis example

```c
int main() {
    char userInput[100];
    getInput(userInput);
    processInput(userInput);
    return 0;
}
```

```c
void getInput(char *input) {
    printf("Enter a command: ");
    fgets(input, 100, stdin);
    // Remove newline character if present
    input[strcspn(input, "\n")] = '\0';
}
```

source

```c
void processInput(char *input) {
    char command[200];
    // Unsafe concatenation (vulnerable to command injection)
    snprintf(command, sizeof(command), "echo %s", input);
    executeCommand(command);
}
```

```c
void executeCommand(char *command) {
    printf("Executing command: %s\n", command);
    system(command); // Unsafe use of system() with tainted input
}
```

Taint analysis example

```c
int main() {
    char userInput[100];
    getInput(userInput);
    processInput(userInput);
    return 0;
}
```

```c
void getInput(char *input) {
    printf("Enter a command: ");
    fgets(input, 100, stdin);
    // Remove newline character if present
    input[strcspn(input, "\n")] = '\0';
}
```

source

```c
void processInput(char *input) {
    char command[200];
    // Unsafe concatenation (vulnerable to command injection)
    snprintf(command, sizeof(command), "echo %s", input);
    executeCommand(command);
}
```

taint propagation

```c
void executeCommand(char *command) {
    printf("Executing command: %s\n", command);
    system(command); // Unsafe use of system() with tainted input
}
```

Taint analysis example

```c
int main() {
    char userInput[100];
    getInput(userInput);
    processInput(userInput);
    return 0;
}
```

```c
void getInput(char *input) {
    printf("Enter a command: ");
    fgets(input, 100, stdin);
    // Remove newline character if present
    input[strcspn(input, "\n")] = '\0';
}
```

source

```c
void processInput(char *input) {
    char command[200];
    // Unsafe concatenation (vulnerable to command injection)
    snprintf(command, sizeof(command), "echo %s", input);
    executeCommand(command);
}
```

taint propagation

```c
void executeCommand(char *command) {
    printf("Executing command: %s\n", command);
    system(command); // Unsafe use of system() with tainted input
}
```

sink

Symbolic Execution

- simulates program execution
- treat program variables as symbols
- symbolic program execution state as a triple
  - cs: control state (next statement, like Program Counter in CPUs)
  - σ: variables state
  - π: path predicate (variable constraints to reach cs)
- we can check if a statement is reachable by checking π feasibility
  - using a Satisfiability Modulo Theory (SMT) solver
  - extension of SAT (satisfiability problems)
    - check if a system of Boolean expressions has a solution
    - extension for other data types (arithmetic, arrays, etc.)
  - e.g., Microsoft Z3
- example tool: KLEE
  - http://klee-se.org

```
1  int process_data(int x, int y, int z) {
2      if(x>10)
3          if(y<20)
4          {
5              int i = y - z;
6              if(z>30)
7                  return 1;
8              else if(i<0)
9                  return -1;
10             else
11                 return 0;
12         }
13     return 0;
14 }
15
```

cs = {7}

$\sigma = \{x=x_0, y=y_0, z=z_0, i=y_0-z_0\}$

$\pi = x>10 \ \&\& \ y<20 \ \&\& \ z>30 \ \&\& \ i<10$

cs = {7}

$\sigma = \{x=x_0, y=y_0, z=z_0, i=y_0-z_0\}$

$\pi = x>10 \ \&\& \ y<20 \ \&\& \ z>30 \ \&\& \ y-z<10$

unsatisfiable

Concolic Execution

- hybrid technique mixing
  - **conc**rete execution (we need to give some inputs)
  - symb**olic** execution
- (partially) solves the problem of having too complex SMT problems
  - e.g. non-linear conditions
- we run the program (or a portion of it) both concretely and symbolically
  - by giving concrete values to variables, SMT problems become simpler
  - symbolic execution can generate inputs to test all the program
    - automatic generation of test cases to cover all possible control flows
- example tool: angr
  - https://angr.io
  - let's play with it ☺

Concolic Execution

- hybrid technique mixing
  - **conc**rete execution (we need to give some inputs)
  - symb**olic** execution
- (partially) solves the problem of having too complex SMT problems
  - e.g. non-linear conditions
- we run the program (or a portion of it) both concretely and symbolically
  - by giving concrete values to variables, SMT problems become simpler
  - symbolic execution can generate inputs to test all the program
    - automatic generation of test cases to cover all possible control flows
- example tool: angr
  - https://angr.io
  - let's play with it ☺

Issues with static analysis

- complex data flows
  - dynamic memory allocation, pointer arithmetic
- external/system libraries
- performance overhead on large applications
- multi-threading
  - e.g. race conditions

Dynamic analysis: fuzzy testing

- idea: random and/or invalid input can trigger a bug!
- aka fuzz testing or fuzzing
- requires a lot of iterations
  - automatic generation of test cases
  - fuzzer = automatic input generator
  - CPU intensive
  - memory intensive (a lot of logs)
- fuzzing does not guarantee a bug-free application!
- first fuzzer created in 1988 by Barton Miller
- first network fuzzer created in 1999 (PROTOS by Oulu University)

Fuzzy testing work-flow

1. study the format
2. fuzz some data
3. send the data to the application
4. monitor for “something strange”
5. if an error occurred
   - find it
   - fix it
1. repeat

Fuzzing approaches

- different (partially overlapping) categories
  - depending on the knowledge of the program to fuzz
    - white-box: full knowledge of the program to fuzz
    - grey-box: partial knowledge, e.g., no data structure but some static analysis data
    - black-box: no knowledge at all
- depending on the input they generate
  - generation-based: generate inputs from scratch
  - mutational: need samples of valid inputs then work on them
  - model-based: formal representation of the inputs
- depending on the complexity of transformations
  - dumb: execute generic input transformations
  - smart: use abstraction and other analysis tool outputs to generate new valid inputs

Fuzzing inputs generation

- generation-based fuzzer
  - generates the input from scratch
  - e.g., random fuzzing: generate random data
    - easy to configure
    - does not depend on the existence or quality of a corpus of seed inputs
    - low coverage
    - can spot hidden bugs or stress in a badly written program
- mutational fuzzing: start from a valid input (seed) and mutate it to generate a new
  - e.g., give some inputs (seeds), then the fuzzer generates new ones
  - easy to configure
  - may reach good coverage
    - quality of seeds affects the coverage

Fuzzing inputs generation

- model-based, grammar-based, or protocol-based
  - model must be explicitly provided
    - may not be available when the software is proprietary
  - harder to configure
    - too much effort to set up
  - excellent coverage

What can I fuzz?

- files
  - textual files: JSON, HTML, configuration files, ...
  - binary files: JPEG, MP4, ...
- network traffic
  - the fuzzer can be the client or the server
  - simple protocols: IP, TCP, UDP, …
  - complex protocols: HTTP, QUIC, ...
- but I need something else (e.g. a string)!
  - find a fuzzer that can do this
  - write your own fuzzer
  - convert a file into what I need

What bugs can I discover?

- usually
  - crashes
  - memory related errors
  - hangs
  - race conditions
- useful tools
  - external tools (e.g. debuggers, …)
  - sanitizers
- regression testing
  - comparison w.r.t. a working copy

How well have I tested my application?

- ideally: check my application states
- problem: very hard to to
- code coverage
  - check if a source line has been executed or not
  - much easier
  - supported by several tools (e.g. gcov)

Commercial fuzzing frameworks

- Synopsis Defensics: https://www.synopsys.com
- Peach Fuzzer: https://www.peach.tech/
- beSTORM: https://www.beyondsecurity.com
  - only for network traffic

Open-source fuzzing frameworks

- abnfuzzer: https://github.com/nradov/abnffuzzer
- AFL: http://lcamtuf.coredump.cx/afl/
- AFL++: https://aflplus.plus
- boofuzz: https://github.com/jtpereyda/boofuzz
- fuddly: https://github.com/k0retux/fuddly
- honggfuzz: https://github.com/google/honggfuzz
- MozPeach: https://github.com/MozillaSecurity/peach
- Radamsa: https://github.com/aoh/radamsa

Symbol table & family

- map between addresses and names
- GCC flag -g: add debugging information
- the developer friend
  - make debugging much easier
  - peek symbol table: nm <binary>
  - peek debug info: readelf --debug-dump <binary>
- the attacker friend
  - make code understanding much easier
  - remove the symbol table & debug info!
    - use GCC flag -s
    - use strip -s <binary>
  - beware of libraries and plug-in frameworks

GDB

- very powerful Linux debugger
- easier to use with a GUI
- use via CLI: gdb --args <binary> <arguments>
- some useful commands
  - r: run the application
  - b <file>:<line>: place breakpoint
  - d <file>:<line>: delete breakpoint
  - backtrace: print stack trace
  - c: continue execution
  - q: quit

GCC: code hardening (I)

- Lubarsky's Law of Cybernetic Entomology
  - There is always one more bug
- buffer overflows (BOF) & co. are common
  - very dangerous!
  - can be used for nefarious actions
  - can be hard to detect
- compiler: automatic code hardening
  - detect stealth bugs
  - make attacks harder
  - slow the performance a bit
  - many options by GCC and LLVM
  - the compiler is your friend!

GCC: code hardening (II)

- -Wall -Wextra -Wformat -Wformat-security
  - enable warnings
- -D_FORTIFY_SOURCE=2
  - enable BOF checks on standard functions
  - if fails use -D_FORTIFY_SOURCE=1
- -fstack-protector-all
  - enable BOF protection
- -pie -fPIE
  - enable address space randomization
- -Wl,-z,now -Wl,-z,relro
  - disable lazy binding
  - make some sections read-only after initialization

Sanitizers (I)

- https://github.com/google/sanitizers
- developed by Google
- supported by GCC and LLVM
- code instrumentation used to detect bugs
- run-time checks
- useful only when testing/debugging
- mostly useful when using GCC flag -g
- slow down the application (1.5-2 times)
- increase the memory usage (3-5 times more)

Sanitizers (II)

- address sanitizer: -fsanitize=address
  - memory access errors
- leak sanitizer: -fsanitize=leak
  - memory leaks
- thread sanitizer: -fsanitize=thread
  - race conditions
- undefined sanitizer: -fsanitize=undefined
  - overflows
  - various arithmetic problems
  - null pointer dereferencing
- some sanitizers are incompatible

Valgrind

- http://valgrind.org
- debugging/profiling suite for C/C++ applications
- various tools
  - memcheck: valgrind <binary>
    - memory access problems
    - memory leaks
  - DRD: valgrind --tool=drd <binary>
    - race conditions
    - deadlocks
- works directly on the binary
- better if you compile with -g
- slow down performance

AFL: American Fuzzy Lop?

- http://lcamtuf.coredump.cx/afl/
- mutational fuzzing of files
- only for C/C++ programs
- found bugs in: Firefox, VLC, iOS kernel,OpenSSL, …
- uses genetic algorithms
  - bit flips
  - addition/subtraction of integers to the bytes
  - insertion of bytes
- basic idea
  1. generate fuzzed file (for maximizing coverage)
  2. run application
  3. compute coverage
  4. repeat

AFL++

- evolution of AFL
  - original AFL project stopped updates in 2017
- supports various source code instrumentation modules
  - LLVM mode, afl-as, GCC plugin
- supports various binary code instrumentation modules
  - meaning you don’t need the source code
  - QEMU mode, Unicorn mode, QBDI mode

AFL: work-flow

1. build: afl-gcc <GCC parameters>
   - do not work with the newer GCC 7.3.x+ versions
   - set AFL_CC to force a compiler
1. fuzz: afl-fuzz -i <dir> -o <logs> <cmd>
   - <dir> = directory containing the seeds
   - <logs> = directory for writing the logs
   - <cmd> = command to launch
     - @@ is replaced by the fuzzed filename
   - AFL_SKIP_CPUFREQ=1 ignore the governor
1. look at the files in <logs>
2. fix the bugs
3. repeat

AFL: logs

- logs/fuzzer_stats: some statistics
  - cycles_done: number of full mutation cycles
  - execs_done: number of executions
  - paths_total: number of paths
  - paths_found: number of executed paths
  - unique_crashes: crashes with different paths
  - unique_hangs: timeouts with different paths
- logs/crashes: fuzzed files producing crashes
- logs/hangs: fuzzed files producing timeouts
- logs/queue: fuzzed files for distinctive paths

AFL: when to stop?

- hard to tell
- ideally: until the entropic death of the Universe
- typical criteria
  - wait at least one full mutation cycle
  - wait until no more paths/bugs are found
  - when my code coverage is high enough
- plots: afl-plot <logs> <dir>
  - <dir> = directory where put some plots
  - useful for finding if we reached “stability”

AFL: advanced crash analysis

- same as before but use afl-fuzz
  - with the flag -C
  - using some files stimulating one type of crash
- AFL will generate file related to only one type of crash
  - comparing them can ease the bug fixing

AFL: corpus minimization

- corpus = set of initial files
- corpus minimization = remove unneeded files
  - useless because they do not increase coverage
  - bad because they slow down the fuzzing
- process
  1. build: with afl-gcc as usual
  2. minimize: afl-cmin -i <dir> -o <min> <cmd>
  - <dir> = directory with the corpus
  - <min> = directory with minimized corpus
  - <cmd> as before
- an interesting idea: fuzz, minimize queue and refuzz

AFL: parallel fuzzing

- AFL is single threaded
- launch separate AFL instances
  - easy, but dumb
  - no synchronization = a lot of identical test cases
- use AFL in parallel mode
  - tricky, but synchronized
  - multiple nohup afl-fuzz copies, but with the flags
    - -M <id>: deterministic mutations
    - -S <id>: random mutations
  - check with afl-whatsup <logs>
  - use kill or killall to stop the fuzzing

AFL: dictionaries

- AFL is (mostly) optimized for compact binary files
- with textual/verbose files AFL is a slow learner
  - some keywords are hard to guess
- you can use a dictionary file to help AFL
  - list of key=”value” pairs
  - the key is actually ignored
- call afl-fuzz with -x <dictionary>

GCOV?

- code coverage is useful
  - to check how good are my tests
  - for profiling my application hot spots
- GCOV allows to compute
  - line coverage
  - branch coverage
  - basic block coverage
- only works with GCC

GCOV: work-flow

1. compile with --coverage
   - *.gcno files are generated
2. launch your application
   - *.gcda files are generated
   - launches are cumulative
     - delete a .gcda file to reset
3. launch: gcov <file.c>
   - <file.c>.gcov is generated
   - launch with -b to get the branch coverage

LCOV

- LCOV can generate a nice HTML coverage report
- work-flow
  1. compile & launch as before
  2. lcov -c -o <file> -d <dir>
     - <dir> = directory with the coverage files
     - <file> = report file
  3. genhtml -o <report> <file>
     - <report> = directory with the HTML report
- useful when your project is big
- easier to navigate

boofuzz?

- generational fuzzing of network traffic
- successor of Sulley
- support two type of low-level connections
  - sockets: TCP, UDP, SSL/TLS, L2/L3 protocols
  - serial connections
- FTP and HTTP support included
- write a script in Python

1. describe the protocol
   - the messages (block requests)
   - the messages relationships
1. connect to an address and start fuzzing

boofuzz: work-flow

1. create a connection
    - SocketConnection(<ip>, <port>, proto = <proto>)
    - SerialConnection(<port>, <baudrate>)
2. create a target (or more for parallel fuzzing)
    - Target(<connection>)
3. create a session and add all the targets
    - session = Session()
    - session.add_target(target)
4. describe the protocol
    - set of functions to describe protocol message structure
    - distinguishing between fuzzable parts and keywords/static parts
    - s_initialize, s_string, s_static, s_delim, s_byte, s_word, s_dword
5. describe the starting messages and start fuzzing
    - session.connect(s_get(<name>))
    - session.fuzz()

Evil fuzzing

- fuzzing can also be used as a form of attack
- an input generating a crash is dangerous
  - it can deny a service (for some time)
  - if due to a buffer overflow
    - an attacker can execute arbitrary code
- 0-day exploits: potentially very dangerous
- how to avoid fuzzing attacks? you can't
- but you can mitigate them
  - use a firewall with rate-limiting
  - chroot jails
  - limit process privileges
  - and in the end: use software with fewer bugs

Fuzzing attack work-flow

1. identify a target
2. study the format
3. fuzz some data
4. send the data to the application
5. monitor for “something strange”
6. if an error occurred
    1. detect exploitability
    2. make evil things
7. repeat

Fuzzing attacks: problems (for the attacker)

- target identification
  - who I attack in a network with 1000 services?
  - use scanning tools to explore the network
  - identify the software versions
    - check known vulnerabilities
    - open-source = offline fuzzing
- format identification
  - standard protocol/format = read RFC & co.
- no access to the source = black-box fuzzing
  - work at assembly level
  - no debugging symbols
  - crash are easy to detect, other issues no so easy

Fuzzing 101: tutorial for AFL++

- https://github.com/antonio-morales/Fuzzing101
- Repository with various usage examples of AFL++
  - with real-life vulnerabilities…
  - …and commented solutions
- To install AFL++
  - suggested solution: docker image
  - docker pull aflplusplus/aflplusplus
  - docker run --platform=linux/amd64 -ti -v .:/src aflplusplus/aflplusplus
    - you need –platform=linux/amd64 only if you are on ARM processor (e.g. Mac M1/M2/M3/M4)
  - apt install git
  - git clone https://github.com/antonio-morales/Fuzzing101