CSE 410/565 Computer Security
Spring 2022

Lecture 16: Building Secure Software

Department of Computer Science and Engineering
University at Buffalo

Review

- A large number of software vulnerabilities
  - various types of buffer overflows
  - input injection attacks
  - integer overflow
  - format string problems
  - interaction with environment variables
  - race conditions
- What can we do to improve software security?

Overview

- Defensive programming: what it is and how it is useful
- How can we make software safer?
  - handling program input
  - writing safe code
  - interacting with the environment
  - handling program output

Defensive Programming

- Defensive programming is the practice of defensive software design to ensure that the software performs as expected in adversarial environment
  - the goal is to ensure correct operation in face of unanticipated usage of the software
  - main difference between normal practices and defensive programming is that nothing is assumed
  - any assumptions about the input and interaction with other components of the system are made explicit
    - user input, file contents, network data, database contents, environment variables, libraries, etc.
    - e.g., it is not assumed that function or library calls outside of the program will work as advertised

Defensive Programming

- Defensive programming (cont.)
  - all assumptions are validated and handled in the code
  - all error states are accounted for
- How can it be achieved?
  - assumption validation is performed for the same components as before
    - checking of input and program parameters
    - validation of environment variables, interaction with operating system, etc.

Defensive Programming

- Software security should be a design goal addressed from the start of program development
  - if it's not, the resulting program is unlikely to be secure
  - any assumptions made about the input and/or the environment must be validated in the program
  - any time changes are made to a secure program, the assumptions need to be revisited
  - the need for secure software is not sufficiently recognized
    - time pressure, insufficient funding
- Regular testing techniques won't identify many vulnerabilities triggered by unusual inputs

Creating Secure Code

- Input handling
  - input size, input interpretation, input syntax
  - examples
    - program arguments cannot be trusted including the program name itself
    - program arguments cannot be assumed to be shorter than the maximum length of a command line in shell
  - several languages now include function calls to aid in input validation
    - e.g., PHP has `mysql_real_escape_string()` that escapes special characters in its argument string for use in SQL queries
  - use regular expressions to validate the input

Creating Secure Code

- Input fuzzing
  - is a technique for testing many potential types of abnormal inputs
  - was introduced in 1989 to help anticipate potential problems in a program when used on adversarial inputs
  - the main idea is to use randomly generated data as inputs to a program
  - the range of inputs can be very large
    - use random textual or binary inputs
    - generate random network requests
    - pass random parameters to functions

Input Fuzzing

- Example of input fuzzing
  - standard HTTP GET request
    - GET /index.html HTTP/1.1
  - anomalous requests
    - AAAAA...AAAA /index.html HTTP/1.1
    - GET ///////////index.html HTTP/1.1
    - GET %n%n%n%n%n.html HTTP/1.1
    - GET /AAAAAAAAAAAAAAA.html HTTP/1.1
    - GET /index.html HTTTTTTTTTTTP/1.1
    - GET /index.html HTTP/1.1.1.1.1.1.1

Input Fuzzing

- Regression vs. Fuzzing
  - regression prescribes running program on many normal inputs, looks for badness
    - the goal is to prevent normal users from encountering errors (i.e., assertions are bad)
  - fuzzing prescribes running program on many abnormal inputs, looks for badness
    - the goal is to prevent attackers from encountering exploitable errors (i.e., assertions are often ok)
- There are several types of fuzzing
  - black-box fuzz testing
  - constraint-based automatic test case generation

Input Fuzzing

- Black box fuzz testing
  - given a program, simply feed it random inputs to see whether it would crash
  - advantages: really easy
  - disadvantages: inefficient
    - only a very small fraction of inputs triggers a crash, probability of running across them might be low
    - input often requires structure, random inputs are likely to be malformed
  - enhancements to the basic approach exist
    - mutation based fuzzing, generation based fuzzing

Input Fuzzing

- Mutation-based black-box fuzzing
  - take a well-formed input, randomly perturb it (by flipping bits, etc.)
  - little or no knowledge of input structure is assumed
  - introduced anomalies can be completely random or follow some heuristics
    - e.g., remove NULL, shift characters, etc.
  - existing tools
    - ZZUF (`http://caca.zoy.org/wiki/zzuf`) is very successful in finding bugs in real-world programs
    - Taof, GPF, ProxyFuzz, FileFuzz, etc.

Input Fuzzing

- Example: fuzzing a PDF viewer
  - Google for .pdf (about a billion results)
  - crawl pages to build a corpus
  - use a fuzzing tool or script to take a file and mutate it
    - feed the file to the program and records if it crashes
- Advantages
  - very easy to setup and automate, no protocol knowledge is required
- Disadvantages
  - limited by the initial corpus, may fail for protocols that use checksums, challenge-response, etc.

Input Fuzzing

- Generation-based fuzzing
  - test cases are generated from some description of the format
    - e.g., RFC, documentation, etc.
  - anomalies are added to each possible spot in the inputs
  - knowledge of protocol is expected to give better results than random fuzzing
  - advantages
    - completeness, can deal with complex dependencies such as checksums
  - disadvantages
    - have to have protocol specification, writing generator can be labor intensive

Input Fuzzing

- Existing generation-based fuzzing tools
  - generational fuzzers for common protocols (ftp, http, SNMP, etc.)
    - Mu-4000, Codenomicon, PROTOS, FTPFuzz
  - fuzzing frameworks: you provide a spec, they provide a fuzz set
    - SPIKE, Peach, Sulley
  - dumb fuzzing automated: you provide files or packet traces, they provide fuzz set
    - Filep, Taof, GPF, ProxyFuzz, PeachShark
  - special purpose fuzzers
    - ActiveX, regular expressions, and others

Input Fuzzing

- How much fuzzing is enough?
  - mutation based fuzzers are able of producing an infinite number of test cases, when has the fuzzer run long enough?
  - example
    - I have a 250KB PDF file
    - suppose the program crashes if one specific byte is changed to a particular value
    - you are expected to run hundreds of thousand tests before finding the bug, is that days?
  - code coverage can be used as a metric of how much has been covered and whether more tests are needed
    - coverage data can be obtained using profiling tools such as `gcov`

Input Fuzzing

- Constraint-based automatic test case generation
  - look inside the box: use the code itself to guide fuzzing
  - assert security/safety properties
  - explore different execution paths to check whether the security properties hold
  - challenges
    - for a given path, need to somehow check whether an input can violate the security property
    - find inputs that will go down different execution paths

Input Fuzzing

- Example

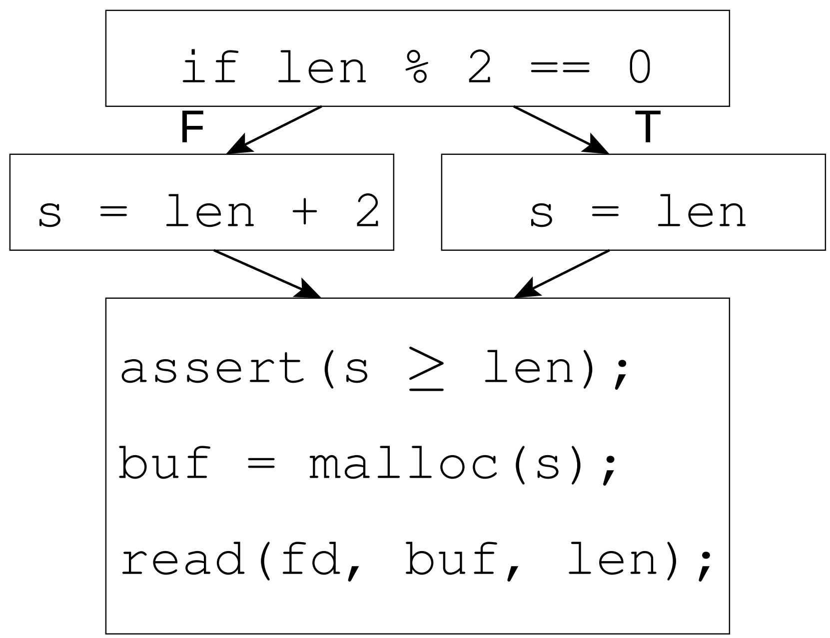
```
func(unsigned int len) {
    unsigned int s;
    char *buf;
    if (len % 2 == 0) s = len;
    else s = len + 2;
    buf = malloc(s);
    read(fd, buf, len);

    ...
```

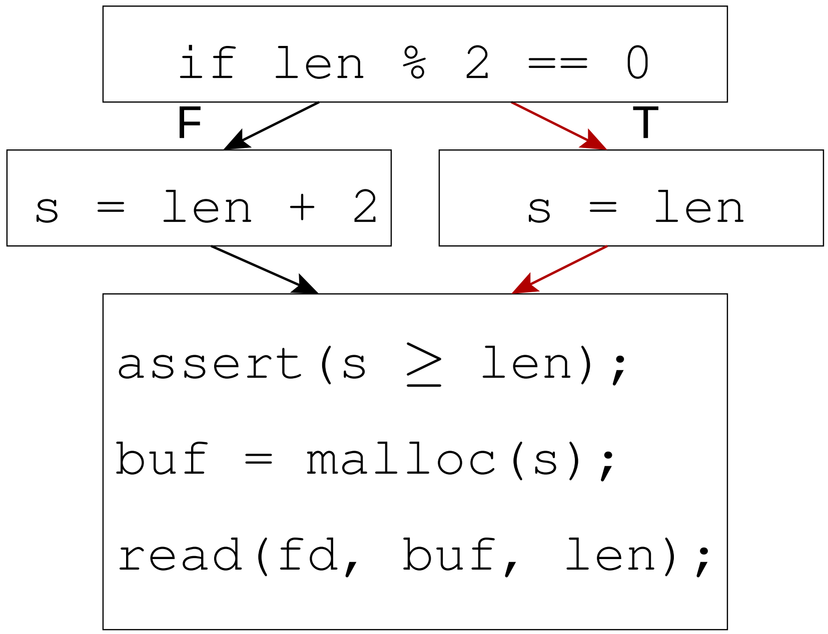
  - where is the bug?
  - what is the security/safety property?
  - what inputs will cause violation of the security property?
  - how likely will random testing find the bug?

Input Fuzzing

- Identify all paths



Input Fuzzing



- Test `len = 8`
  - no assertion failure
  - what about all inputs that take the same path as `len = 8`?

Input Fuzzing

- Solution: symbolic execution
  - represent inputs (i.e., `len`) as symbolic variables
  - perform each operation on symbolic variables symbolically
  - construct a formula for a given path and give it to a solver
  - example
    - is there a value for `len` s.t.
      `len % 2 = 0` ∧ `s = len` ∧ `s < len`?
    - in this case the formula is not satisfiable, the solver returns no
    - this means that for any `len` that follows this path, the execution will be safe
  - symbolic execution can check many inputs at the same time

Input Fuzzing

- Symbolic execution (cont.)
  - how do we check other paths?
  - reverse condition of the branch to go a different path
    - the condition becomes `len % 2 != 0`
    - the formula becomes
      `len % 2 != 0` ∧ `s = len + 2` ∧ `s < len`
  - the solver returns satisfying assignment `len =` $2^{32} - 1$
  - the bug is found
- Some available tools: EXE, DART, CUTE

Creating Safe Code

- Correct implementation is also important to program safety
  - ensure that algorithms are appropriate
    - e.g., a strong pseudo-random number generator is used, all code used in testing has been removed, etc.
    - search for patterns such as “fix”, “assume”, “XXX”, etc.
  - ensure that stored values are interpreted correctly
    - i.e., a memory location is interpreted according to the same data type as what was stored in that memory
    - use pointers with caution
  - ensure correct memory usage
    - freeing memory after use to avoid memory leaks, freeing only after the last use

Creating Safe Code

- Program interaction with the environment
  - carefully check (or don't use) critical environment variables
  - exercise the principle of least privilege
    - use groups for escalated privileges whenever possible
    - grant only necessary privileges (e.g., to a web server)
    - partition a complex program into sub-tasks with appropriate separate privileges
  - handle access to shared resources correctly
    - use atomic operations to obtain exclusive access to a resource
    - e.g., check for a lock file by attempting to create it

Creating Safe Code

- Program interaction with the environment (cont.)
  - exercise safe temporary file use
    - use unpredictable temporary file names
    - handle file creation operation with care or use atomic operations
    - grant minimum access privileges on temporary files
  - be aware of operating system interactions and optimizations
    - securely deleting a file is an excellent example of how the program might not perform as expected due to OS optimizations
      - are the data being written to the original data blocks?
      - are the data being repeatedly written?

Creating Safe Code

- Program interaction with the environment (cont.)
  - verify interaction with other programs for correctness
    - inputs passed from another program should not be assumed trusted (or having common origin)
    - check exit status of child processes
    - use suitable data protection for network-based communication
- Handling program output
  - use correct encoding
  - apply necessary protection

Summary

- Writing safe code is an extremely non-trivial task
  - explicitly validate all assumptions about program input and environment
  - use safe programming practices
  - use any tools and techniques for testing that resources permit
    - code review, static analysis, fuzzing, . . .