CS165 – Computer Security

Vulnerability discovery & static analysis

Nov 4, 2021

Our Goal

– How to exploit a vulnerability?
– How to find them?

Our Goal

- How to exploit a vulnerability? ✓
- How to find them?

Vulnerability

- How do you define computer "vulnerability"?
  - *Flaw*
  - *Accessible to adversary*
  - *Adversary has ability to exploit*

Vulnerability

- How do you define computer "vulnerability"?
  - *Flaw – Can we find flaws in source code?*
  - *Accessible to adversary – Can we find what is accessible*
  - *Adversary has ability to exploit – Can we find how to exploit?*

Vulnerability

- How do you define computer "vulnerability"?
  - *Flaw – Can we find flaws in source code?*
  - *Accessible to adversary – Can we find what is accessible (attack surface)?*
  - *Adversary has ability to exploit – Can we find how to exploit?*

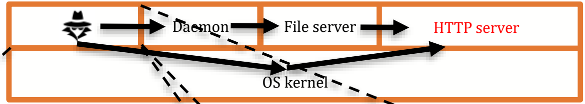
Vulnerability

- How do you define computer "vulnerability"?
  - *Flaw – Can we find flaws in source code?*
  - *Accessible to adversary – Can we find what is accessible (attack surface)?*
  - *Adversary has ability to exploit – Can we find how to exploit (highly dependent on types of flaws, e.g., buffer overflow)?*

One Approach

- Run the program on various **low-integrity** inputs (i.e., through identified **attack surface**)
  - See what happens
  - Maybe you will find a flaw
- How should you choose inputs?

Analyzing Attack Surface



Sequence of syscalls

Number	Name	Description
1	exit	terminate process execution
2	fork	fork a child process
3	read	read data from a file or socket
4	write	write data to a file or socket
5	open	open a file or socket
6	close	close a file or socket
37	kill	send a kill signal
90	old_mmap	map memory
91	munmap	unmap memory
301	socket	create a socket
303	connect	connect a socket

Key	Value
Req type	1
Payload	"abc"
…	…

Dynamic Analysis Options

- Regression Testing
  - Run program on many normal inputs and look for bad behavior in the responses
    - Typically looking for behavior that differs from expected – e.g., a previous version of the program
- Fuzz Testing
  - Run program on many abnormal inputs and look for bad behavior in the responses
    - Looking for behaviors that may be triggered by adversaries
      - Bad behaviors are typically crashes caused by memory errors

Dynamic Analysis Options

- Which approach is more likely to find vulnerabilities?
- Why?

Fuzz Testing

- Fuzz Testing
  - Idea proposed by Bart Miller at Wisconsin in 1988
- Problem: People assumed that utility programs could correctly process any input values
  - Available to all
- Found that they could crash 25-33% of UNIX utility programs

Fuzz Testing

- Fuzz Testing
  - Idea proposed by Bart Miller at Wisconsin in 1988
- Approach
  - Generate random inputs
  - Run lots of programs using random inputs
  - Identify crashes of these programs
  - Correlate with the random inputs that caused the crashes
- Problems: Not checking returns, Array indices…

```
inp=`perl -e '{print "A"x8000}'`
for program in /usr/bin/*; do
  for opt in {a..z} {A..Z}; do
    timeout -s 9 1s $program -$opt $inp
  done
done
```

1009 Linux programs. 13 minutes. 52 *new* bugs in 29 programs.

Example

```
#include <string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

Along as the input is longer than 64, there will be a bug found!

Challenges

- Idea: Search for possibly accessible and exploitable flaws in a program by running the program under a variety of inputs
- Challenge: Selecting input values for the program
  - What should be the goals in choosing input values for dynamic analysis?

Challenges

- Idea: Search for possibly accessible and exploitable flaws in a program by running the program under a variety of inputs
- Challenge: Selecting input values for the program
  - What should be the goals in choosing input values for dynamic analysis?
    - *Find all exploitable flaws*
    - *With the fewest possible inputs*
- How should these goals impact input choices?

Black Box Fuzzing

- Like Miller – Feed the program random inputs and see if it crashes
- Pros: Easy to configure
- Cons: May not search efficiently
  - May re-run the same path over again (low coverage)
  - May be very hard to generate inputs for certain paths (checksums, hashes, restrictive conditions)
  - May cause the program to terminate for logical reasons – fail format checks and stop

Black Box Fuzzing

- Example

```
function( int type, char *buf )
{
  if ( type == MAGIC_NUMBER1) {
     if ( check_format( buf )) {
        update( buf );
     }
  }
}
```

Mutation-Based Fuzzing

- Supply a well-formed input
  - Generate random changes to that input
- No assumptions about input
  - Only assumes that variants of well-formed input may problematic
- Example: zzuf
  - http://sam.zoy.org/zzuf/
  - Reading: The Fuzzing Project Tutorial
    - https://fuzzing-project.org/tutorials.html

Mutation-Based Fuzzing

- Example: zzuf
  - http://sam.zoy.org/zzuf/
- The Fuzzing Project Tutorial
  - zzuf -s 0:1000000 -c -C 0 -q -T 3 objdump -x win9x.exe
  - Fuzzes the program `objdump` using the sample input `win9x.exe`
  - Try 1M seed values (-s) from command line (-c) and keep running if crashed (-C 0) with timeout (-T 3)

Mutation-Based Fuzzing

- Easy to setup, and not dependent on program details
- But may be strongly biased by the initial input
- Still prone to some problems
  - May re-run the same path over again (same test)
  - May be very hard to generate inputs for certain paths (checksums, hashes, restrictive conditions)

Generation-Based Fuzzing

- Generational fuzzer generate inputs “from scratch” rather than using an initial input and mutating
- However, to overcome problems of naïve fuzzers they often need a format or protocol spec to start
  - GET /blabla HTTP/1.0

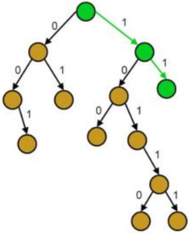
    {Opcode: [3-5 bytes] [space] [n bytes] [4 bytes of characters] … }
  - Examples fuzzers include
    - SPIKE, Peach Fuzz
- However format-aware fuzzing is cumbersome, because you'll need a fuzzer specification for every input format you are fuzzing

Generation-Based Fuzzing

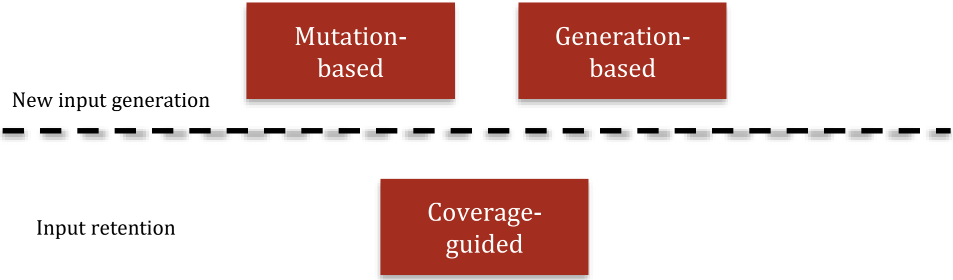
- Can be more accurate, but at a cost
- Pros: More complete search
  - Values more specific to the program operation
  - Can account for dependencies between inputs
- Cons: More work
  - Get the specification
  - Write the generator – ad hoc
- Need to do for each program

Grey Box Fuzzing

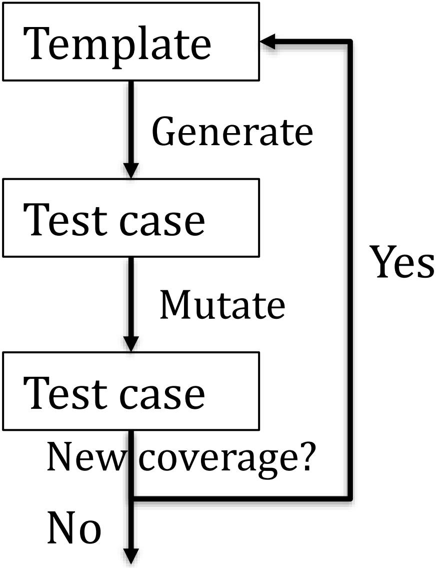
- Rather than treating the program as a black box, instrument the program to track the paths run
  - Also called coverage-guided
- Save inputs that lead to new paths
  - Associated with the paths they exercise
- Example
  - American Fuzzy Lop (AFL)
- "State of the practice" at this time



Relationship of fuzzing techniques



Syzkaller




```
# Copyright 2015 syzkaller project authors. All rights reserved.
# Use of this source code is governed by Apache 2 LICENSE that can be founc

include <linux/kvm.h>
include <linux/kvm_host.h>
include <uapi/linux/fcntl.h>
include <asm/kvm.h>
include <asm/mce.h>

resource fd_kvm[fd]
resource fd_kvmvm[fd]
resource fd_kvmcpu[fd]
resource fd_kvmdev[fd]
resource fd_sgx_provision[fd]

openat$kvm(fd const[AT_FDCWD], file ptr[in, string["/dev/kvm"]], flags flag
openat$sgx_provision(fd const[AT_FDCWD], file ptr[in, string["/dev/sgx_prov

ioctl$KVM_CREATE_VM(fd fd_kvm, cmd const[KVM_CREATE_VM], type const[0]) fd_
ioctl$KVM_GET_MSR_INDEX_LIST(fd fd_kvm, cmd const[KVM_GET_MSR_INDEX_LIST],
ioctl$KVM_CHECK_EXTENSION(fd fd_kvm, cmd const[KVM_CHECK_EXTENSION], arg ir
ioctl$KVM_GET_VCPU_MMAP_SIZE(fd fd_kvm, cmd const[KVM_GET_VCPU_MMAP_SIZE])
```

Sample template

AFL

- Provides compiler wrappers for gcc to instrument target program to collect fuzzing stats

AFL

- Provides compiler wrappers for gcc to instrument target program to collect fuzzing stats
- See
  - http://lcamtuf.coredump.cx/afl/

AFL Build

- Provides compiler wrappers for gcc to instrument target program to collect fuzzing stats
- Replace the gcc compiler in your build process with afl-gcc
- For example, in the Makefile

```
CC=path-to/afl-gcc
```

- Then build your target program with afl-gcc
  - Generates a binary instrumented for AFL fuzzing

AFL Use

- Provides compiler wrappers for gcc to instrument target program to collect fuzzing stats
- Run the fuzzer using afl-fuzz

```
afl-fuzz –i <input-dir> -o <output-dir> <path-to-bin> [args]
```

- `output` is the directory where the AFL results will be placed

AFL Demo

AFL Output

- Shows the results of the fuzzer
  - E.g., provides inputs that will cause the crash
- File “fuzzer_stats” provides summary of stats – UI
- File “plot_data” shows the progress of fuzzer
- Directory “queue” shows inputs that led to paths
- Directory “crashes” contains input that caused crash
- Directory “hangs” contains input that caused hang

AFL Operation

- How does AFL work?
  - http://lcamtuf.coredump.cx/afl/technical_details.txt
- Fuzzing strategies
  - Highly deterministic at first – bit flips, add/sub integer values, and choose interesting integer values
  - Then, non-deterministic choices – insertions, deletions, and combinations of test cases

Grey Box Fuzzing

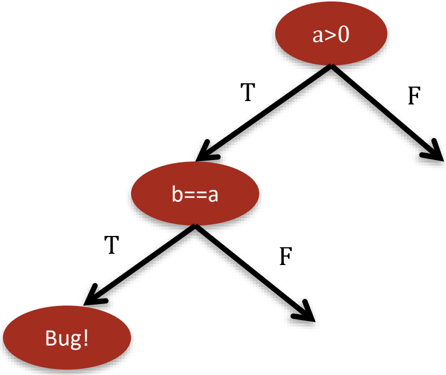
- Finds flaws, but still does not understand the program
- Pros: Much better than black box testing
  - Essentially no configuration
  - Lots of crashes have been identified
- Cons: Still a bit of a stab in the dark
  - May not be able to execute some paths
  - Searches for inputs independently from the program
- Need to improve the effectiveness further

White Box Fuzzing

- Combines test generation with fuzzing
  - Test generation based on static analysis and/or symbolic execution – more later
  - Rather than generating new inputs and hoping that they enable a new path to be executed, compute inputs that will execute a desired path
    - And use them as fuzzing inputs
- Goal: Given program with a set of input parameters, generate a set of inputs that maximizes code coverage

Symbolic execution

```
void func(int a) {
    if(a > 0)  {
        if(b == a) {
            // bug is here
        }
    }
}
```



$a>0 \land b==a \xrightarrow{\text{SMT solver}} a=1, b=1$

Take Away

- Goal is to discover vulnerabilities in our programs before adversaries exploit them
- One approach is dynamic testing of the program
  - Fuzz testing aims to achieve good program coverage with little effort for the programmer
  - Challenge is to generate the right inputs
- Black box (Mutational and generation), Grey box, and White box approaches are being investigated
  - AFL (Grey box) is now commonly used

Dynamic Analysis Limits

- Major advantage
  - When we produce a crash, it is a real crash
- Issue
  - However, may not be exploitable (i.e., not a vulnerability)
  - On the other hand, want to fix memory errors
    - But often not assertion failures
- Major limitation
  - We cannot find all vulnerabilities in a program
- Why not?

Questions