ROP and Fuzzing

Agenda

Return Oriented Programming (ROP)

Fuzzing

Return Oriented Programming (ROP)

Recall: Data Execution Prevention (DEP), aka NX

- Memory can be either “data” or “code” and only “code” can run
- Attackers usually load their malicious programs as data
- DEP prevents data from being executed

ret2libc

We can't execute data because of DEP.

Instead, execute existing code with malicious parameters:

- E.g. invoke bash with "rm –rf"

Problems with this for an attacker:

- Can only use predefined functions
- Can only invoke a single function per exploit
- Available functions and their effects vary on different systems
- Dangerous commands have been intentionally removed to stop this

Attacker's Goal

Execute arbitrary code…

Using only code that is already on the box…

Without requiring specific complete functions to be on the box.

Tools

There is lots of predictable code on every box.

- OS files
- Common programs
- The program containing the vulnerability you're exploiting

Almost any short snippet of assembly can be found inside these programs.

How could we link these snippets together?

Return Oriented Programming (ROP)

Use "return" statements to chain together short snippets of existing programs.

- Ret statement = pop+jmp = go to the location stored on top of the stack
- "Gadgets": a few useful statements followed by "ret"
  - E.g.:

```
Add eax, 1
ret
```

- "ROP chain": a set of gadgets that makes a program

Exploitation

Overrun the vulnerable buffer with your ROP chain on the stack

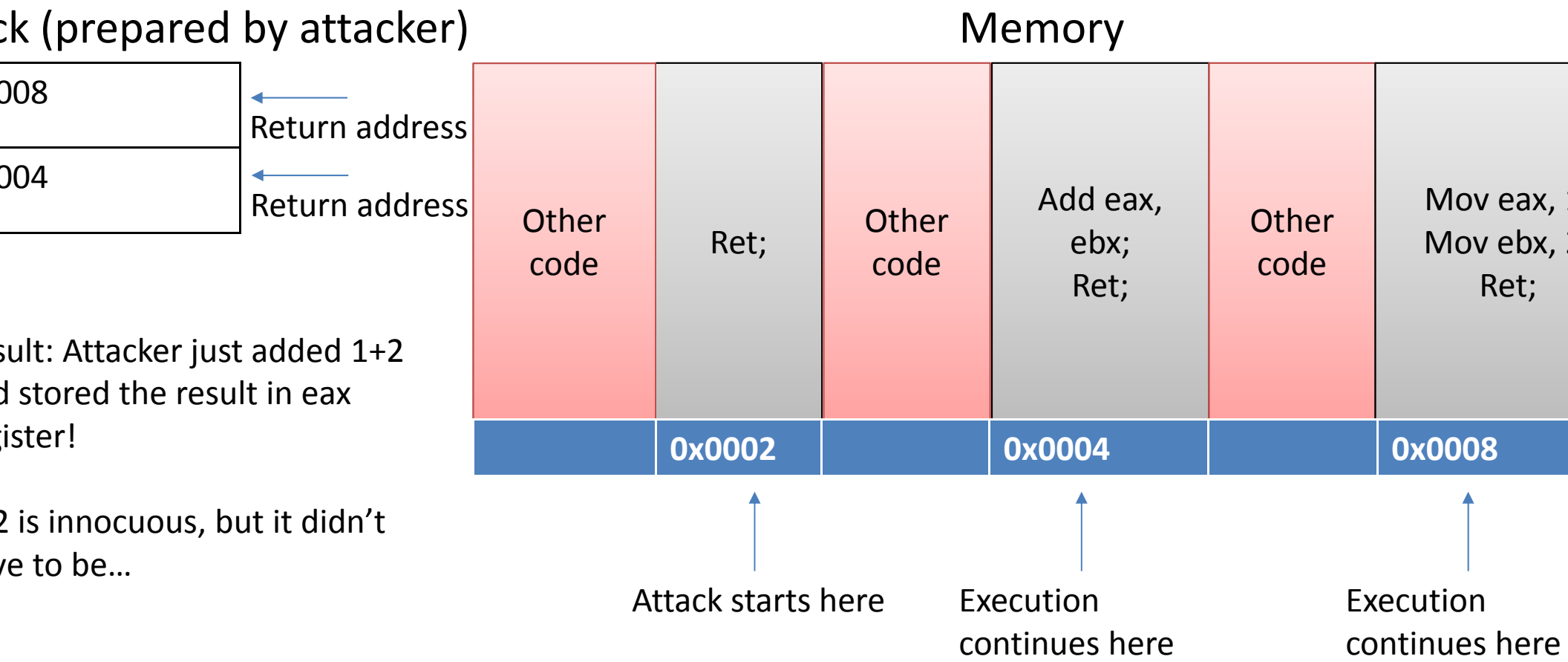
Trigger a return instruction, either directly or by reaching the end of the vulnerable function

Each gadget pointed to by the ROP chain executes in sequence

Attacker code executes

Your bank account becomes a money telegram in Elbonia

ROP Chain


ck (prepared by attacker)
008
004
Return address
Return address
sult: Attacker just added 1+2
d stored the result in eax
gister!
2 is innocuous, but it didn't
ve to be…
Memory
Other code
Ret;
Other code
Add eax, ebx; Ret;
Other code
Mov eax,
Mov ebx,
Ret;
0x0002
0x0004
0x0008
Attack starts here
Execution continues here
Execution continues here

Loading the Stack

Easy!

We're exploiting code in some function

- Call it drawPicture(a, b, c)

As arguments, a, b and c are on the stack

a, b, and c are attributes of the picture

The picture and its attributes come from the document

The document comes from the attacker

Therefore the arguments on the stack come from the attacker

Wrinkles

You may not control the entire stack

But the right gadget will pop the extraneous data off the stac

E.g.: capital letters are part of the ROP chain, lowercase aren’

Stack:
A
b
C
D
E
Etc.

A:
Pop;
ret;

C:
Useful stuff
Etc.

Even more gadgets

- Gadgets can be pieced together from other parts of the program:
  - String literals:
    - "hello world" has some meaning if interpreted as opcodes
    - It's probably gibberish and unexecutable (=crash)
    - But perhaps "hel" has a useful meaning as opcodes
      - » Or "llo", or "rld", or …
  - Jump into middle of multi-byte instructions
    - E.g.:
      - » 0F 1F = nop;
      - » 1F = pop;
    - So we can turn nop into pop

Automated Gadgets

Identify DLLs that are likely to be on the target system

- We'll call it target.dll
- Hopefully it has ASLR off

Write a normal program and compile it into exploit.dll

For each machine-language instruction in exploit.dll:

A. Scan target.dll to find that instruction followed by "ret"
B. Store the offset of that location: this is the next memory location i
   your ROP chain.

Example

Malicious code:

```
Stos eax, 1;
Add eax, 1;
Call sys_function;
```

ROP chain:

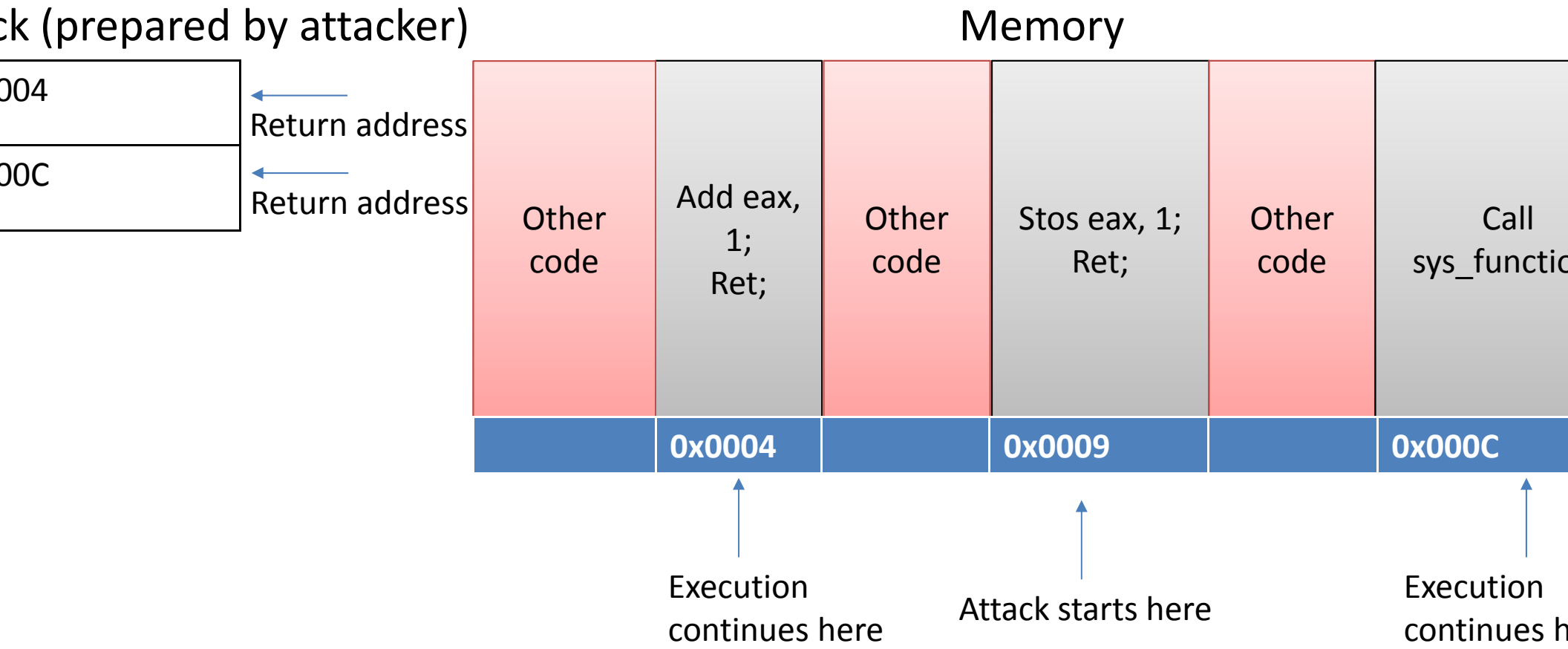
9

4

12

0: …
1: …
2: …
3: …
4: add eax, 1;
5: ret;
6: …
7: …
8: …
9: stos eax, 1;
10: ret;
11: …
12: call sys_function;

Example during exploit



Mitigating ROP

Hard! The attack only uses real code.

ASLR

Patching: ROP requires a specific version of a DLL/EXE

Possibilities:

- Dynamic instrumentation
- Intra-function randomization of statement order
- Protect ret instructions from improper use (Onarlioglu, et al., 2010)
- Shadow stack (many papers)
- Control flow integrity

Dynamic Instrumentation

Monitor every instruction using a separate program

Use antivirus-like methods to detect “unusual” behavior of processes

- Signatures
- Heuristics
- Machine learning

In practice, too slow and unreliable to be usable in most case

Useful for screening suspected malware

Intra-Function Randomization

These two functions are identical:

```
Add eax, 1;          Add ebx, 1;
Add ebx, 1;          Add eax, 1;
Ret;                 Ret;
```

If a gadget expects “add eax” and gets “add ebx”, it won’t wo

To break ROP, randomly reorder functions like this at runtime

Only partially successful – not all gadgets can be reordered

But, works at runtime = available today in EMET

Protect ret instructions

Put a cookie before every ret

Slow, hard to implement

Too many ret-like instructions

- Fixed-offset call
- Pop+jmp
- Etc.

Shadow Stack

- Create a copy of the stack
- Before returning, check to see if the copy matches the origina
  in key attributes like return address
- Totally works
- Requires getting a new CPU
- Doesn’t play nicely with multicore (sync issues/race condition
- Uses lots of RAM
- And various other problems

Control flow integrity

Many variations on this theme with various properties

Map valid entry and exit points at compile/link, verify at run time

Problems:

- Requires recompiling/relinking existing software
- Doesn't work well for DLLs

Tricks to get around these problems either kill performance o leave security holes

- Some hope here for hardware-based measures in the future

Putting it all together: real-world exploits

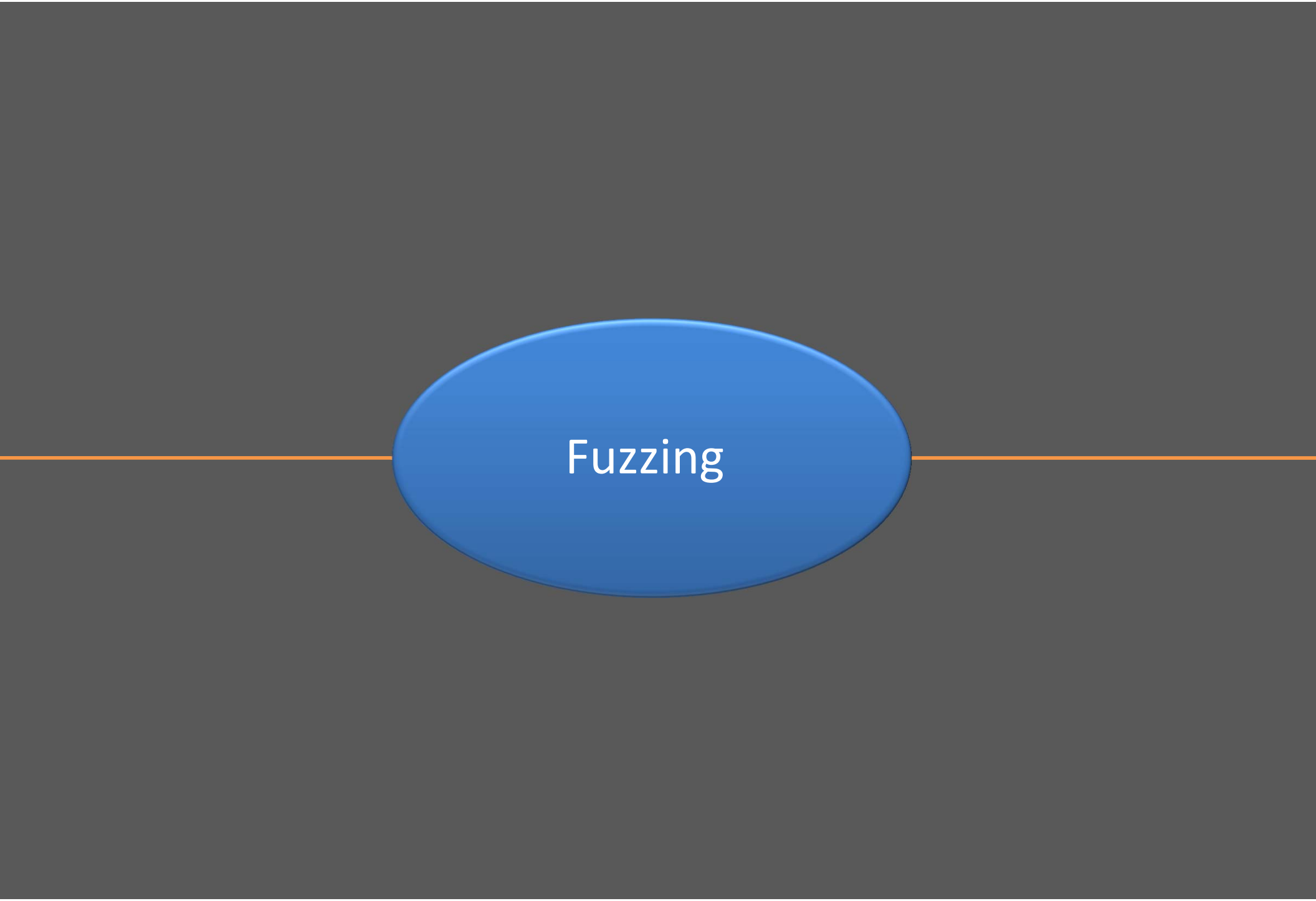
Exploit vulnerability: load stack, start ROP chain

ROP into a non-ASLR binary, usually an ActiveX control

- The ROP chain disables DEP and implements step 3

Heap spray to fill memory with NOP sled, payload at end

Last step in ROP chain is a jump into the NOP sled


Fuzzing

Ethics: Coordinated Disclosure

It's easy to hurt real people by doing what I'm about to describe.

If you test a production (live) system, you might:

- Cause a server outage (denial of service)
- Expose others' personal data

If you find a vulnerability and expose its details to others before the software maker has a chance to fix it, you might cause people to get attacked, lose money, even lose their live

There are consequences for these things.

Basic Idea

Automatically find vulnerabilities using randomly generated input.

Most security vulnerabilities happen when a program mishandles malformed input.

Let's give the program a lot of malformed input and see what happens!

Procedure

Start with a legitimate file

Randomly corrupt it

Open it with the target program

Did it crash?

- If so, it might be a vulnerability! Save the corrupted file and crash dump
- If not, just discard it.

Goto 1

Pros and Cons of Fuzzing

Pros:

- Very easy to set up
- Fully automated and scalable: more CPU cycles = more bugs
- Doesn’t require source code or debug symbols

Cons:

- Effectiveness limited by templates
- False positives: most crashes aren’t vulnerabilities
- Compression, encryption, checksums must be worked around

In practice, one of the main ways vulnerabilities are found

Types of Fuzzing

Dumb: no awareness of underlying format

- Random or sequential

Smart: aware of underlying format

- Target specific parts of a format that you believe to be vulnerable
- Fix checksums
- Decompress, fuzz, recompress

Generative vs. mutative: build files from scratch vs. modify existing files

- Generative is best for simple formats like TCP

Dumb Mutative Fuzzing Example

```
hile true
   For currentByte from bytes in inFile
        If randomFloat()>0.2 ← A parameter of the fuzzer
             randomChar() >> outfile
        else
             currentByte >> outfile
   End
   run “target.exe outfile”
   if crashed(target.exe)
        save crash dump, outfile
```

Smart Mutative Fuzzing Example

```
hile true
   For xmlNodes from nodes in parseXML(inFile)
        If xmlNode.name == “FuzzingTarget” &&
                                        randomFloat()>0.2
             xmlNode.Value = randomInt
             xmlNode >> outfile
        else
             xmlNode >> outfile
   End
   run “target.exe outfile”
   if crashed(target.exe)
        save crash dump, outfile
```

Dumb Generative Fuzzing Example

```
dev/rand | tcp
```

ote: dumb generative fuzzing doesn’t work very well)

Smart Generative Fuzzing Example

```
hile true
   For xmlNodes from nodes in
   nodesInFileFormat
            xmlNode.Value = randomInt
            xmlNode >> outfile
   End
   run “target.exe outfile”
   if crashed(target.exe)
        save crash dump, outfile
```

Smart vs. Dumb Tradeoffs

Smart fuzzers:

- Harder to write
- Slower per repetition
- Only works for one format
- Find more bugs per rep
- Have the word “smart” in their name, sounds good when you tell your boss what you’re working on

Charlie Miller wrote a dumb fuzzer in 5 lines of Python and found dozens of bugs and got paid to speak at BlackHat about it. So…

Smart Fuzzers Sometimes Required

Smart fuzzing required for:

- Encoded formats, e.g. .docx/.xlsx/.pptx (.zip with XML inside)
- Checksums
- XML

Dumb fuzzing these files is only fuzzing the zip/XML parser

- Worth doing, but not as good as fuzzing the full file load pathway

Smart fuzzer would unzip, fuzz files, re-zip

- And only fuzz the data fields within the XML, not the tags

Other tradeoffs in fuzzing

Non-security crashes and graceful failures can hide security bugs

- Your document has 5 fuzzed fields
- Fuzzed field #4 will cause a security bug
- Fuzzed field #2 will cause a non-security crash
- Result: non-security crash, you don't find the security bug

More complicated templates:

- Better code coverage = more chance to find bugs
- But also more non-security crashes

Lots of changes per rep: better chance to find bugs, but more chance of hitting a non-security crash too

- If you work for Microsoft, you just make the team fix the non-security crashes too so you don't have to worry about this ☺

What is an “exploitable” crash?

One that subverts the control flow of the program

Typically:

- Almost all WriteAVs
- ReadAVs that fetch data that is used as the target of a jump
- ReadAVs that fetch data that is used in a conditional expression

Usually not/never:

- Null ReadAVs
- ReadAVs where the data is not used to control program flow
- Exceptions
- Divide by zero

Why?

Virtually always, the condition that caused a crash is controlled by the attacker

WriteAV = the attacker controls where data is written = the attacker can overwrite anything he wants including return addresses

ReadAV where the data is used as the target of a jump = the attacker controls where the program jumps next

ReadAVs that fetch data that is used in a conditional expression = the attacker controls which codepath the program takes next

- Not always exploitable

Why Not?

Null ReadAVs, divide by zero: instant crash

ReadAVs where the data is not used to control program flow: some data might get corrupted, but there's rarely a way to tu this into control of the CPU

Exceptions:

- A designed, controlled way to handle errors
- Only a crash to the user
- To the system, it's a "graceful shutdown with no UI"

Sifting through the debris

Debugger plugins automate triage of crashes

!exploitable: WinDBG plugin

- Renders verdicts like “EXPLOITABLE”, “NOT_EXPLOITABLE”…
- And also lots of “PROBABLY EXPLOITABLE” and “UNKNOWN”

Valgrind: Linux open source debugger with exploitability analysis tools

These aren’t foolproof but dramatically improve ROI

Real-world Fuzzers

Custom scripts/programs (most common)

Peach: Highly customizable general-purpose fuzzer

MiniFuzz: Very simple fuzzer from Microsoft

Why doesn't fuzzing find all the bugs?

Probabilistic

Code coverage: if your fuzzer+templates don't hit a code path you'll never fund bugs in that code path

- E.g. if your Word templates have no pictures, you'll never find bugs the picture loading code

Only finds memory corruption bugs

Some exploitable bugs are hidden by non-exploitable bugs

Some bugs required complex conditions to hit

Also…

It only works if you do it

- For every program
  - For every file format/protocol that program supports

And then fix all the bugs you find.

That sounds like a lot of work.

What to do if you find a vulnerability

If you think you've found a vulnerability in real software:

- Write up everything you've found, including:
  - The file that causes the crash and any information about where the fault lie
  - How you found the bug
  - Where in the program it appears to be crashing (stack trace)
  - Why you think it's exploitable (!exploitable output)
- Send it to the maker's published security contact email address
- Work with them to get it fixed
- Put it on your resume and give me a call