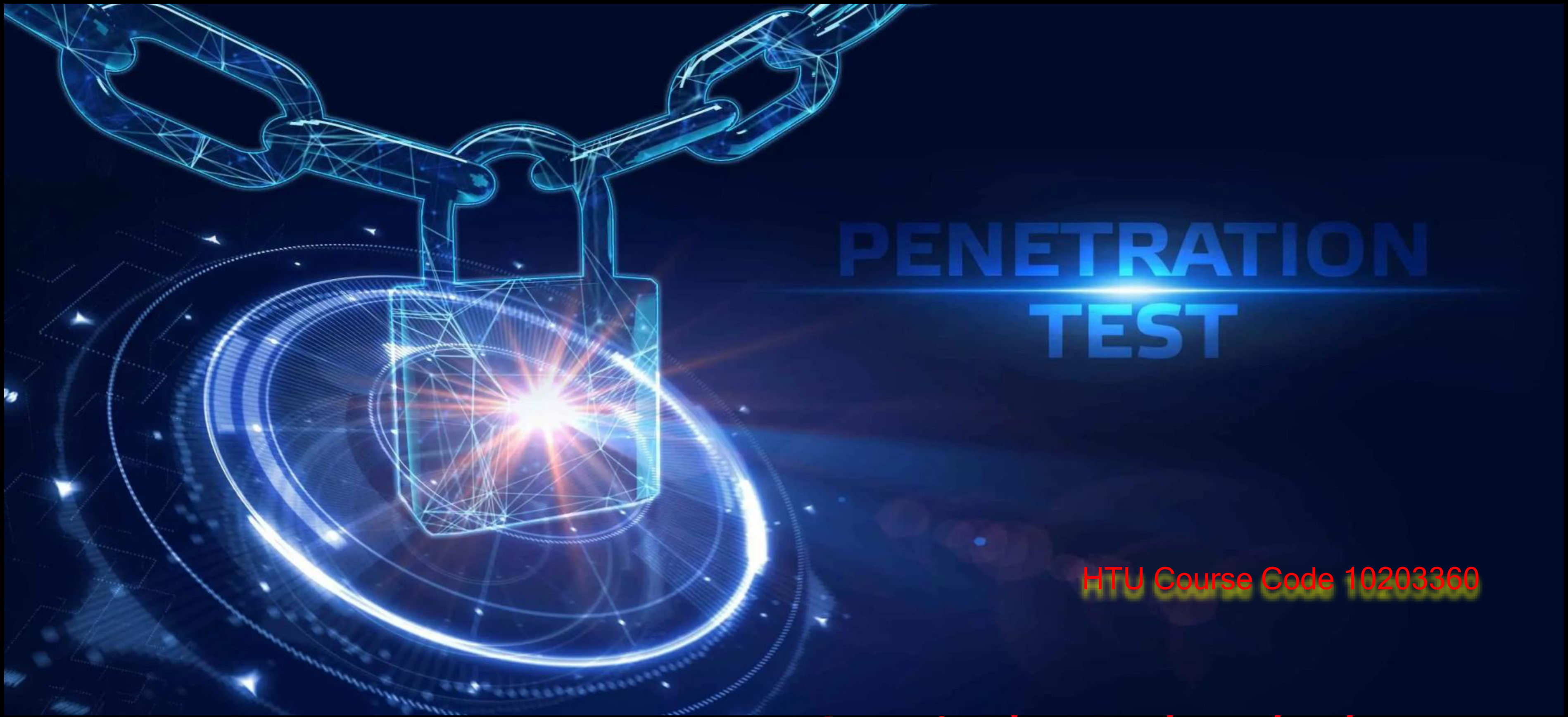


HTU Course Code 10203360

By: Sami Al-mashaqbeh

LEARNING OUTCOMES

LO#3: Gain proficiency in advanced exploit development techniques by mastering essential debugging skills, understanding memory corruption vulnerabilities, and exploring strategies for bypassing exploit mitigations.

- Discuss in detail the differences between static and dynamic code analysis, and their importance in penetration testing.

- Ability to use fuzzers and debuggers like Immunity Debugger to analyze software running in memory and successfully fuzz it to determine ways to crash the software.

Vulnerability Identification and Fuzzing

a quick road to bug hunting …

Bug Hunting

- Bug hunting is the process of finding bugs in software or Hardware.
- Security bugs (aka software security vulnerabilities and security holes) allows attackers to:
  - Remotely compromise systems
  - Escalate local privileges
  - Cross privilege boundaries
  - Wreak havoc on a system!

4 Fun & Profit

- Finding security bugs was done for fun and to get media attention
- Today, organizations are paying for security researchers to identify bugs
  - Bounty programs (Google, FaceBook, Twitter, RedHat, etc)
  - Zero Day Initiative (ZDI)
  - iDefense
  - Tipping Point
  - Pwn2Own
  - Others? Please add

Taking Advantages of Bugs

- Software that take the advantages of a software vulnerability are called “exploits”
- Exploiting a widely used application, os, protocol, etc. will grab huge media coverage and attention
  - Road to become a Hacking Star

Exploits Language

- No specific language for writing exploits
- Exploits can be written using any programming language
    - C, C++, Perl, JavaScript, Assembly, and PYTHON !
- I prefer Python for it's simplicity and for the huge range of libraries that could be used for creating a PoC or working exploit

Bug Hunting Formal Process

- Writing software is a *human art*, and two different coders may code the same function with the same requirements differently!
- For that reason IMO, Bug Hunting is a *human art* too!
- No formal process to finding bugs in SW, but there are a couple of techniques that can be used for bug discovery

Common Techniques

- Static Analysis
  - Static Code Analysis
  - Reverse Engineering
- Dynamic Analysis
  - Debugging
  - Fuzzing

- Each technique has it's pros and cons
  - Bug hunters mix it up

Static Analysis

- Static Code Analysis
  - Code is needed
  - Tedious and time consuming
  - Requires high knowledge and/or skills with given language
  - Costs a lot (expensive)
- Reverse Engineering
  - Code not needed
  - Requires the binary file
  - Time consuming
  - High technical skill is needed (assembly!)

Dynamic Analysis

- In this lecture we will be covering
  - Debugging and Fuzzing

General Bug Hunting Methodology

Understand the Application

- Read specs / documentation
  - understand purpose or business logic
- Examine attack surface
  - inputs, configuration
- Identify target components an attacker would hit
  - think like an attacker to defend better:
    - try to hit the Database for SQLi?
    - try to upload a file?
    - try to spawn a shell?

What Leads to Bugs?

- Miscalculations
- Failure to validate input
- Programmer failure to understand an API
- Failure to validate results: operations, functions, etc
- Application state failures
- Complex protocols
- Complex file formats
- Complex encoding / decoding / expansion
- etc

Debugging

Debugger

- A computer program that lets you run your program, line by line and examine the values of variables or look at values passed into functions and let you figure out why it isn't running the way you expected it to.

Why use Debuggers

- Debuggers offer sophisticated functions such as:
  - Running a program step by step (single-stepping mode),
  - Stopping (breaking) (pausing the program to examine the current state) at some event or specified instruction by means of a breakpoint,
  - Tracking the values of variables,
  - Tracking the values of CPU registers,
  - Attach to a process,
  - View the process's Memory map,
  - Load memory dump (post-mortem debugging),
  - Disassemble program instructions,
  - Change values at runtime,
  - Continue execution at a different location in the program to bypass a crash or logical error.

Common Debuggers

- GNU Debugger (GDB)
- Microsoft Windows Debugger (Windbg)
- OllyDbg
- Immunity Debugger
  - Based on Ollydbg
- Microsoft Visual Studio Debugger
- Interactive DisAssembler (IDA Pro)

Common Debugger   Cont.

- Ollydbg
  - Most popular for malware analysis
  - User-mode debugging only
  - IDA Pro has a built-in debugger, but it's not as easy to use or powerful as Ollydbg

- Windbg
  - Supports kernel-mode debugging

Disassembler v. Debuggers

- A disassembler like IDA Pro shows the state of the program just before execution begins
- Debuggers show
  - Every memory location
  - Register
  - Argument to every function at any point during processing
    - And let you change them

Source Level v.s Assemby Level Debuggers

- Source-level debugger
  - Usually built into development platform
  - Can set breakpoints (which stop at lines of code)
  - Can step through program one line at a time

- Assembly-level debuggers (low-level)
  - Operate on assembly code rather than source code
  - Malware analysts are usually forced to use them, because they don't have source code

Kernel vs User Mode Debugging

User Mode Debugging

- Debugger runs on the same system as the code being analyzed
- Debugging a single executable
- Separated from other executables by the OS

Kernel Mode Debugging

- Requires two computers, because there is only one kernel per computer
- If the kernel is at a breakpoint, the system stops
- One computer runs the code being debugged
- Other computer runs the debugger
- OS must be configured to allow kernel debugging
- Two machines must be connected

Using a Debugger

Two Ways for Debugging

- Start the program with the debugger
  - It stops running immediately prior to the execution of its entry point
- Attach a debugger to a program that is already running
  - All its threads are paused
  - Useful to debug a process that is affected by malware

Singl Stepping

- Simple, but slow
- Don't get bogged down in details

Example

- This code decodes the string with XOR

Example 9-1. Stepping through code

```
mov      edi, DWORD_00406904
mov      ecx, 0x0d
LOC_040106B2
xor      [edi], 0x9C
inc      edi
loopw    LOC_040106B2
...
DWORD:00406904:    F8FDF3D01
```

Example 9-2. Single-stepping through a section of code to see how it changes memory

```
D0F3FDF8 D0F5FEEE FDEEE5DD 9C (.............)
4CF3FDF8 D0F5FEEE FDEEE5DD 9C (L............)
4C6FFDF8 D0F5FEEE FDEEE5DD 9C (Lo...........)
4C6F61F8 D0F5FEEE FDEEE5DD 9C (Loa..........)
. . . SNIP . . .
4C6F6164 4C696272 61727941 00 (LoadLibraryA.)
```

a

Stepping Over vs Stepping Into

- Single step executes one instruction
- **Step-over** call instructions
  - Bypasses the call
  - Decreases the amount of code you need to analyze
  - Might miss important functionality, especially if the function never returns
- **Step-into** a call
  - Moves into the function and stops at its first command

Pausing Execution with Breakpoints

- A program that is paused at a **breakpoint** is called **broken**
- Example
  - You can't tell where this call is going
  - Set a breakpoint at the call and see what's in eax

Example 9-3. Call to EAX

```
00401008    mov     ecx, [ebp+arg_0]
0040100B    mov     eax, [edx]
0040100D    call    eax
```

- This code calculates a filename and then creates the file
- Set a breakpoint at CreateFileW and look at the stack to see the filename

Example 9-4. Using a debugger to determine a filename

```
0040100B  xor     eax, esp
0040100D  mov     [esp+0D0h+var_4], eax
00401014  mov     eax, edx
00401016  mov     [esp+0D0h+NumberOfBytesWritten], 0
0040101D  add     eax, 0FFFFFFFEh
00401020  mov     cx, [eax+2]
00401024  add     eax, 2
00401027  test    cx, cx
0040102A  jnz     short loc_401020
0040102C  mov     ecx, dword ptr ds:a_txt ; ".txt"
00401032  push    0                ; hTemplateFile
00401034  push    0                ; dwFlagsAndAttributes
00401036  push    2                ; dwCreationDisposition
00401038  mov     [eax], ecx
0040103A  mov     ecx, dword ptr ds:a_txt+4
00401040  push    0                ; lpSecurityAttributes
00401042  push    0                ; dwShareMode
00401044  mov     [eax+4], ecx
00401047  mov     cx, word ptr ds:a_txt+8
0040104E  push    0                ; dwDesiredAccess
00401050  push    edx              ; lpFileName
00401051  mov     [eax+8], cx
00401055 1call    CreateFileW ; CreateFileW(x,x,x,x,x,x,x)
```

a

WinDbg

Figure 9-1. Using a breakpoint to see the parameters to a function call. We set a breakpoint on `CreateFileW` and then examine the first parameter of the stack.

Encrypted Data

- Suppose malware sends encrypted network data
- Set a breakpoint before the data is encrypted and view it

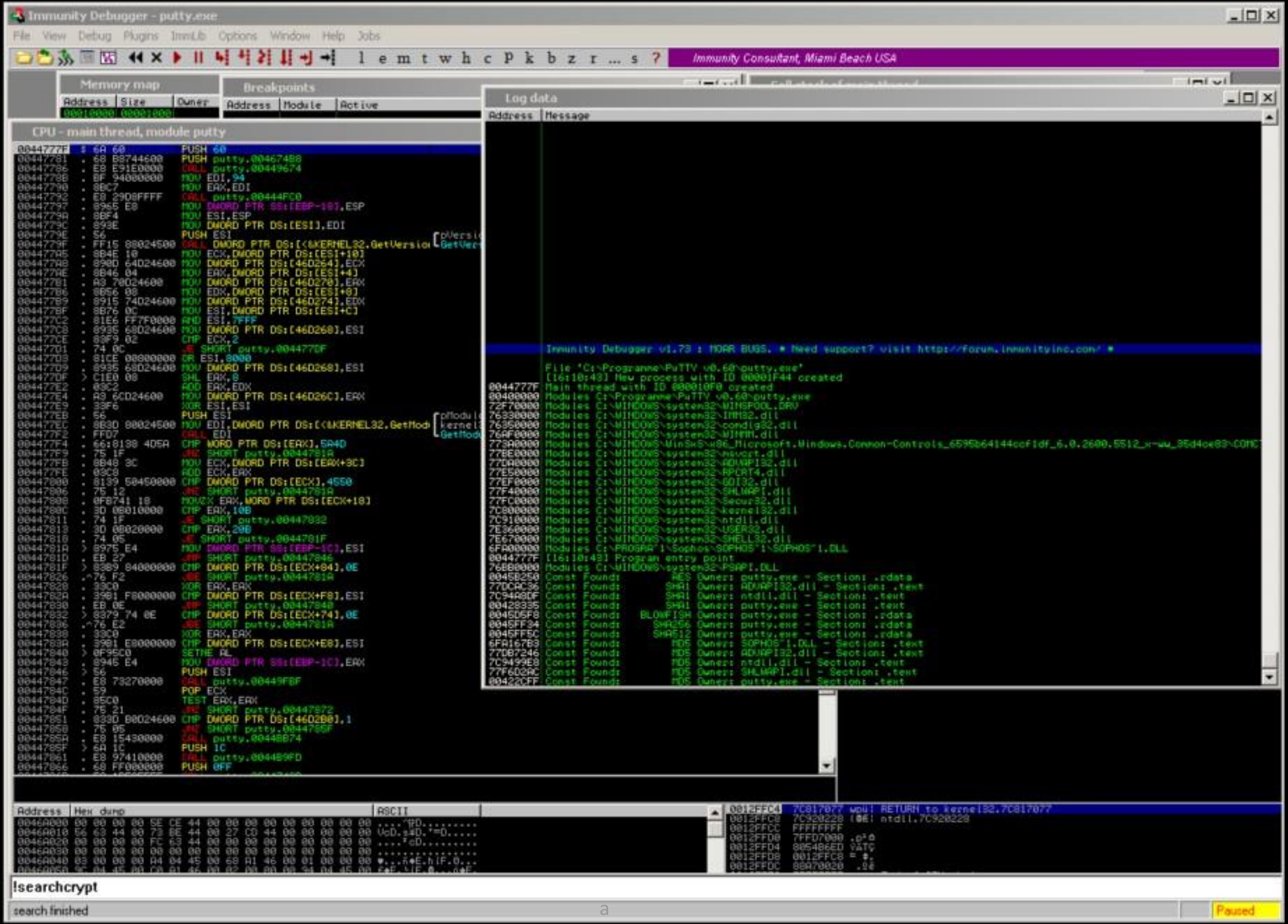
OllyDbg

Figure 9-2. Viewing program data prior to the encryption function call

Immunity Debugger

- A powerful new way to write exploits, analyze malware, and reverse engineer binary files
- It builds on a solid user interface with function graphing, and a large and well supported Python API for easy extensibility

Did you read that? **Python**


Immunity Debugger - putty.exe
File View Debug Plugins ImmLib Options Window Help Jobs
Immunity Consultant, Miami Beach USA
Memory map
Breakpoints
Log data
CPU - main thread, module putty
Immunity Debugger v1.73 : MOAR BUGS. * Need support? visit http://forum.immunityinc.com/ *
!searchcrypt
search finished
Paused

Types of Breakpoints

- Software execution
- Hardware execution
- Conditional

Software Execution Breakpoints

- The default option for most debuggers
- Debugger overwrites the first byte of the instruction with 0xCC
  - The instruction for INT 3
  - An interrupt designed for use with debuggers
  - When the breakpoint is executed, the OS generates an exception and transfers control to the debugger

Memory Contents at a Breakpoint

- There's a breakpoint at the push instruction
- Debugger says it's 0x55, but it's really 0xCC

Table 9-1. Disassembly and Memory Dump of a Function with a Breakpoint Set

Disassembly view				Memory dump
00401130 55	1 push	ebp	00401130	2 CC 8B EC 83
00401131 8B EC	mov	ebp, esp	00401134	E4 F8 81 EC
00401133 83 E4 F8	and	esp, 0FFFFFFF8h	00401138	A4 03 00 00
00401136 81 EC A4 03 00 00	sub	esp, 3A4h	0040113C	A1 00 30 40
0040113C A1 00 30 40 00	mov	eax, dword_403000	00401140	00

When Software Execution Breakpoints Fail

- If the 0xCC byte is changed during code execution, the breakpoint won't occur
- If other code reads the memory containing the breakpoint, it will read 0xCC instead of the original byte
- Code that verifies integrity will notice the discrepancy

Hardware Execution Breakpoints

- Uses four hardware Debug Registers
  - DR0 through DR3 - addresses of breakpoints
  - DR7 stores control information
- The address to stop at is in a register
- Can break on access or execution
  - Can set to break on read, write, or both
- No change in code bytes

Hardware Execution Breakpoints

- Running code can change the DR registers, to interfere with debuggers
- General Detect flag in DR7
  - Causes a breakpoint prior to any mov instruction that would change the contents of a Debug Register
  - Does not detect other instructions, however

Conditional Breakpoints

- Breaks only if a condition is true
  - Ex: Set a breakpoint on the GetProcAddress function
  - Only if parameter being passed in is RegSetValue
- Implemented as software breakpoints
  - The debugger always receives the break
  - If the condition is not met, it resumes execution without alerting the user

Conditional Breakpoints

- Conditional breakpoints take much longer than ordinary instructions
- A conditional breakpoint on a frequently-accessed instruction can slow a program down
- Sometimes so much that it never finishes

Exceptions

Exceptions

- Used by debuggers to gain control of a running program
- Breakpoints generate exceptions
- Exceptions are also caused by
  - Invalid memory access
  - Division by zero
  - Other conditions

First and Second Chance Exceptions

- When a exception occurs while a debugger is attached
  - The program stops executing
  - The debugger is given **first chance** at control
  - Debugger can either handle the exception, or pass it on to the program
  - If it's passed on, the program's exception handler takes it

Second Chance

- If the application doesn't handle the exception
- The debugger is given a **second chance** to handle it
  - This means the program would have crashed if the debugger were not attached
- In malware analysis, first-chance exceptions can usually be ignored
- Second-chance exceptions cannot be ignored
  - They usually mean that the malware doesn't like the environment in which it is running

Common Exceptions

- INT 3 (Software breakpoint)
- Single-stepping in a debugger is implemented as an exception
  - If the **trap flag** in the flags register is set
  - The processor executes one instruction and then generates an exception
- Memory-access violation exception
  - Code tries to access a location that it cannot access, either because the address is invalid or because of access-control protections

Common Exception   Cont.

- Violating Privilege Rules
  - Attempt to execute privileged instruction with outside privileged mode
  - In other words, attempt to execute a kernel mode instruction in user mode
  - Or, attempt to execute Ring 0 instruction from Ring 3

THQ

- Can we modify executables using a debugger?
- Write an example showing howto modify a Windows EXE file.
    - For example bypass a whole check routine or set of instructions!!!

Fuzzing

what garbage data can your application handle?

Fuzzing

- Original research name “Boundary Value Analysis”
- “ n automated method for discovering faults in software by providing unexpected input and monitoring for exceptions.” - Fuzzing

- Also said:

"Fuzzing is the process of sending intentionally invalid data to a product in the hopes of triggering an error condition or fault. These error conditions can lead to exploitable vulnerabilities.“

HD Moore (MSF Founder)

Plz note

- Fuzzing has no rules!
- Not always successful!

Fuzzing History

- Fuzzing is not new
  - It’s been named for about 20 years.
- Professor Barton Miller
  - Father of Fuzzing
  - Developed fuzz testing with his students at the University of Wisconsin-Madison in 1988/89
  - GOAL: improve UNIX applications
- Since 1999 with PROTOS till date, Fuzzing has managed to discover a wide range of security vulnerabilities* (Check Fuzzing 101 for further history information)

Fuzzing Methods

- Sending Random Data
  - Least Effective
  - Unfortunately, sometimes, code is bad enough for this to work

- Manual Protocol Mutation
  - You are the fuzzer
  - Time consuming, but can be accurate when you have a hunch
  - Web App Pen-Testing

Fuzzing Method Cont.

- Mutation or Brute Force Testing
  - Starts with a valid sample
  - Fuzz each and every byte in the sample

- Automatic Protocol Generation Testing
  - Person needs to understand the protocol
  - Code is written to describe the protocol (a “grammar”)
  - Fuzzer then knows which piece to fuzz, and which to leave alone (SPIKE)

What Data can be Fuzzed?

- Virtually anything!
- Basic types: bit, byte, word, dword, qword
- Common language specific types: strings, structs, arrays
- High level data representations: text, xml

Where can Data be Fuzzed?

Across any security boundary, e.g.:

- An RPC interface on a remote/local machine
- HTTP responses & HTML content served to a browser
- Any file format, e.g. Office document
- Data in a shared section
- Parameters to a system call between user and kernel mode
- HTTP requests sent to a web server
- File system metadata
- ActiveX methods
- Arguments to SUID binaries

What Does Fuzzed Data Consist Of?

- Fuzzing at the type level:
  - Long strings, strings containing special characters, format strings
  - Boundary case byte, word, dword, qword values
  - Random fuzzing of data buffers
- Fuzzing at the sequence level
  - Fuzzing types within sequences
  - Nesting sequences a large number of times
  - Adding and removing sequences
  - Random combinations
- Always record the random seed!!

When to Fuzz?

Fuzzing typically finds implementation flaws, e.g.:

- Memory corruption in native code
  - Stack and heap buffer overflows
  - Un-validated pointer arithmetic (attacker controlled offset)
  - Integer overflows
  - Resource exhaustion (disk, CPU, memory)
- Unhandled exceptions in managed code
  - Format exceptions (e.g. parsing unexpected types)
  - Memory exceptions
  - Null reference exceptions

When to Fuzze Cont.

- Injection in web applications
  - SQL injection against backend database
  - LDAP injection
  - HTML injection (Cross-site scripting)
  - Code injection

Two Approaches

- Dumb (mutational) Fuzzing
- Fuzzer lacks contextual information about data it is manipulating
- May produce totally invalid test
- Up and running fast
- Find simple issues in poor quality code

- Smart (generational) Fuzzing
- Fuzzer is context-aware
  - Can handle relations between entities, e.g. block header lengths, CRCs
- Produces partially well-formed cases test cases
- Time consuming to create
  - What if protocol is proprietary?
- Can find complex issues

Two Approache Cont.

- Which approach is better?
- Depends on:
  - Time: how long to develop and run fuzzer
  - [Security] Code quality of target
  - Amount of validation performed by target
    - Can patch out CRC check to allow dumb fuzzing
  - Complexity of relations between entities in data format
- Don't rule out either!
  - My personal approach: get a dumb fuzzer working first
  - Run it while you work on a smart fuzzer

Determining Exploitability

- This process requires experience of debugging security issues, but some steps can be taken to gain a good idea of how exploitable an issue is...
- Look for any cases where data is written to a controllable address - this is key to controlling code execution and the majority of such conditions will be exploitable
- Verify whether any registers have been overwritten, if they do not contain part data sent from the fuzzer, step back in the disassembly to try and find where the data came from

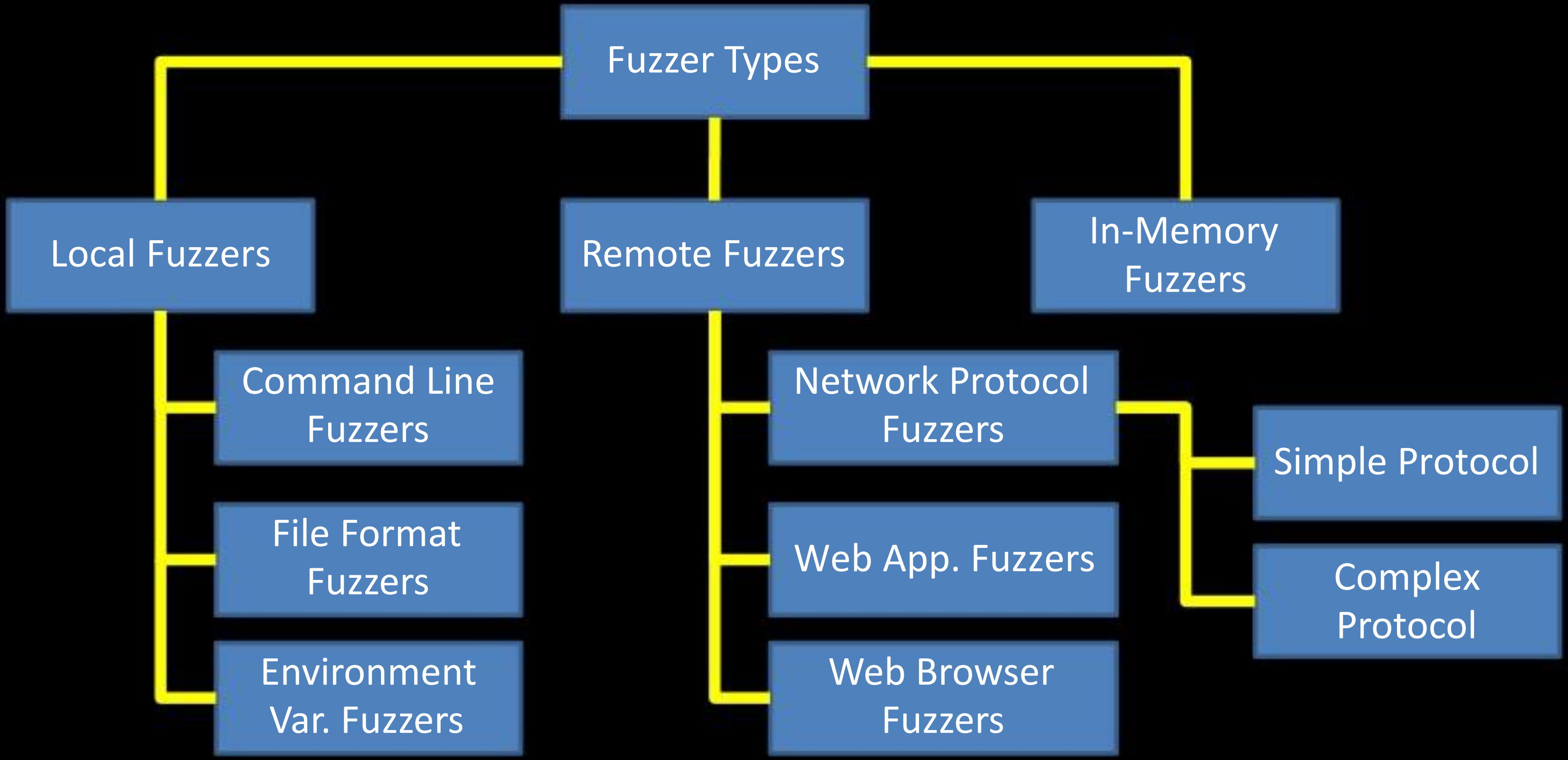
Determining Exploitability Cont.

- If the register data is controllable, point the register which caused the crash to a page of memory which is empty, fill that page with data (e.g., ‘aaaaa…’)
- Repeat and step through each operation, until another crash occurs, reviewing all branch conditions which are controlled by data at the location of the (modified) register to ensure that they are executed

Determining Exploitabilit   Cont.

- Are saved return address/stack variables overwritten?
- Is the crash in a heap management function?
- Are the processor registers derived from data sent by the fuzzer (e.g. 0x61616161)?
- Is the crash triggered by a read operation?
- Can we craft a test case to avoid this?
- Is the crash triggered by a write operation?
- Do we have full or partial control of the faulting address?
- Do we have full or partial control of the written value?

Fuzze Classifications



Types of Fuzzer   Cont.

- File Format Fuzzers
  - Fuzz valid files
  - Pass them to an executable
- Remote Fuzzers (might make you famous)
  - Listen on a network connects
  - When client connects, fuzz them!

Types of Fuzzer Cont.

- Network Protocol Fuzzers
  - The Fuzzer is the client
  - Need to understand the protocol
  - Simple Protocols
    - Text Based: Telnet, FTP, POP, HTTP
  - Complex Protocols
    - Binary Data (some ASCII)
    - Complex authentication, encryption, etc

Types of Fuzzer   Cont.

- Other types of fuzzers:
  - Web Application and Server Fuzzing
  - Web Browser Fuzzing
  - In-Memory Fuzzing

Common Fuzzers

- Publicly available fuzzing frameworks:
  - Spike, Peach Fuzz, Sulley, Schemer, etc
- Publicly available fuzzing applications
  - Fuzz, FileFuzz, iFuzz, WebFuzz, JBroFuzz, WebScarab,
  - BurpSuite (includes a fuzzer), notSPIKEFile, SPIKEProxy, ProtoFuzz
  - SMUDGE, mangleme, FileP, FileH, MalyBuzz,
  - Dfuz, AxMan, bugger, fuzzdb
  - And the list goes on and on

The Fuzzing Process

- Identify Targets
- Identify Inputs
- Generate Fuzzed Data
- Execute Fuzzed Data
- Monitor for Exceptions
- Determine Exploitability

The Fuzzing Process

- Determine Exploitability - Remotely
  - You need to know what data you sent
    - Record all fuzzed strings, making note of exceptions
    - Network Captures (Wireshark)
  - Try and reproduce the scenario
  - Is it a memory corruption bug?
  - Is it an application logic flaw?
- Determine Exploitability - Locally
  - Attach a debugger

Protocol Fuzzing

- Find as much data as you can about the target application
  - Google is your friend
  - Maybe someone has fuzzed it
  - Maybe it uses some standard protocol

- What is the transport layer?
  - TCP or UDP?
    - Effects anomaly detection

- What type of protocol (simple or complex)?

Protocol Fuzzing Cont.

- Do we need to authenticate?
  - What authentication protocol?
- Scoping your assessment
  - You may only care about pre-auth
- Reversing the Protocol
  - Generate Traffic and Sniff
  - Use wireshark (check for plug-ins!)
  - Google
- Once you understand how to communicate with a service, you can send packets to it

Why ???

- Writing a network protocol fuzzer, means eventually you'll be re-inventing the wheel!!!

- Why do that when you can use:

SPIKE

SPIKE

- SPIKE fuzzer released in 2002
  - Written by Dave Aitel (Immunity Inc.)
- SPIKE is a genius
- SPIKE is a fuzzing framework/API
- Ability to describe data
- Built in libraries for known protocols (*RPC)
- Fuzz strings designed to make software fail

SPIK   Cont.

- Simple Text Based Protocol Fuzzing
- ccepts a “script” of SPIKE commands
- Example: ./generic_send_tcp <IP> <PORT> script.spk 00

```
s_readline()
s_string_variable("USER");
s_string(" ");
s_string_variable("devel_user");
s_string(" ");
s_string_variable("PASS");
s_string(" ");
s_string_variable("secretpassword");
s_string("\r\n");
```

SPIKE's Real Value

- Complex Protocols have length fields and data fields
- Tracking length fields while Fuzzing data is complicated
- SPIKE does this for you
- Block Based Protocol Representation

What is a SPIKE?

- “A SPIKE is a simple list of structures which contain block size information and a queue of bytes.”

```
s_block_size_binary_bigendian_word(“somepacketdata”);
s_block_start(“somepacketdata”)
s_binary(“01020304”);
s_block_end(“somepacketdata”);
```

What is a SPIKE Cont.

```
s_block_size_binary_bigendian_word("somepacketdata");
s_block_start("somepacketdata")
s_binary("01020304");
s_block_end("somepacketdata");
```

- Push 4 NULLs onto BYTE queue (size place holder)
- Then a new BLOCK listener is allocated named "somepacketdata"

What is a SPIKE Cont.

```
s_block_size_binary_bigendian_word("somepacketdata");
s_block_start("somepacketdata")
s_binary("01020304");
s_block_end("somepacketdata");
```

- Script starts searching the block listeners for one named "somepacketdata"
- Block "start" pointers are updated to reflect the blocks position in the queue

What is a SPIKE Cont.

```
s_block_size_binary_bigendian_word("somepacketdata");
s_block_start("somepacketdata")
s_binary("01020304");
s_block_end("somepacketdata");
```

- 4 bytes of data are pushed onto the queue

What is a SPIKE   Cont.

```
s_block_size_binary_bigendian_word("somepacketdata");
s_block_start("somepacketdata")
s_binary("01020304");
s_block_end("somepacketdata");
```

- The block is ended, and the sizes are finalized
- The original 4 null bytes are updated with the appropriate size value

What is a SPIKE Cont.

s_block_size_binary_bigendian_word(“somepacketdata”)

s_block_start(“somepacketdata”)

s_binary(“01020304”)

s_block_end(“somepacketdata”)

Block 2 Morepacketdata Big Endian word Start Pointer: 1008	Block 1 Somepacketdata Big Endian word Start Pointer: 1000

Existing Challenges

- How to measure effectiveness of a fuzzer?
  - Number of test cases?
  - Number of bugs?
  - Severity of bugs?
  - % Code coverage?
- How many test cases to run?
  - How to balance complexity vs. time constraints?

Another Spike Example (HTTP)

- Consider the following HTTP Request:

```
POST /admin/login.php HTTP/1.1
Host: www.example.com
Connection: close
User-Agent: Mozilla/6.0
Content-Length: 29
Content-Type: application/x-www-form-encoded

user=admin&password=secret
```

Another Spike (HTTP Cont.

```
s_string("POST /admin/login.php HTTP/1.1\r\n");
s_string("Host: www.example.com\r\n);
s_string("Connection: close\r\n");
s_string("User-Agent: Mozilla/6.0\r\n");
s_string("Content-Length: ");
s_blocksize_string("post", 7);
s_string("\r\nContent-Type: application/x-www-form-encoded\r\n\r\n");
s_block_start("post_args");
s_string("user=");
s_string_variable(“admin");
s_string("&password=");
s_string_variable(“secret");
s_block_end("post");
```

Final Tip to Beginners

- OS: Windows XP first!
  - Easy to debugging
  - Almost every RE tool works on XP
  - For example use Kali or BackTrack for developing tools
- Find bugs and debug/exploit them upon XP
- And port it for other versions of OS (Windows 7,8, etc)
- Virtualization Software (VMWARE, Virtualbox, etc) is mandatory
- Use snapshots
- Your OS will be messed up by your Fuzzer

Language Again LOL

- Don’t listen to others
- Just choose one, whatever you like
- But if you still need a recommendation? Python is my answer
- Many hack libraries are written in python
- If you use C for fuzzing because you just want to feel you’re l33t, you’re wrong!
- Realize what is your goal

BoF Demo

JMP ESP