HOW TO GET TO HERE
FROM HERE

FX Phenoelit

Halvar

TESO

LESA scene at
Agenda

- The Mindset
- Finding vulnerabilities
- Writing exploits
- Exploiting non-standard stuff
Mindset

1. Understand John von Neumann!
   - For any given computer today, there is no difference between data and code.
   - Where the instruction pointer points to, this is code
   - Where other registers point to, this is data
   - Code can become data
   - Data can become code
Open your eyes!

- Cheer when things crash
- Don‘t click away Dr. Watson
- Don‘t just delete ,core‘

Example:

You view a web page with your favorite browser and it crashes.
Warning

- Make no mistake, this is **work**
- Learning as you go on
- Finding vulnerabilities takes time
- Reliable exploitation needs a lot of testing
- Brain required
Finding Vulnerabilities
The goal: 0day

- What we are looking for:
  - Handles network side input
  - Runs on a remote system
  - Is complex enough to potentially contain significant number of vulnerabilities
Testing methods

- Manual testing
- Fuzzing
- Static analysis
- Diff and BinDiff
- Runtime analysis
Manual Testing

- Using the standard client (or server) to access the target service
- Observing the behavior:
  - States in the target
  - Reaction to valid input
  - Reaction to invalid input
  - Information transmitted before and after authentication
  - Default configuration and misconfiguration issues and environment requirements
  - Logging capabilities
Manual Testing

- Things to look for:
  - Input validation on client side
    - Input in client rejected
    - Input in client accepted but modified before transmission
  - Pre-Authentication client data
    - Hostname
    - Username
    - Certificate content
    - Date/Time strings
    - Version information (Application, OS)
Manual Testing

- More things to look for:
  - Network protocol structure
    - Dynamic or static field sizes
    - Field size determination
    - Information grouping
    - Numeric 32bit fields
  - Timing
    - Concurrent connections
    - Fast sequential connections
Manual Testing

- **Advantages:**
  - No need for additional tools
  - Becoming familiar with the target
  - Uncovers client side security quickly
  - Easy correlation between user action and network traffic

- **Disadvantages:**
  - Potentially high learning effort
  - Often provides only clues where vulnerabilities might be found
  - Proving a vulnerability often requires additional efforts (such as code)
  - High dependence on the tester
Manual Testing

- Usual findings:
  - Cross Site Scripting / Code & SQL injections
  - Protocol based overflows and integer issues
  - Application logic failures

- Best suited for:
  - Web applications
  - Java application frameworks
  - Proprietary clients
  - Internet Explorer (and other browsers)
Fuzzing

- Creating rough clients (or counterparts) to generate a wide range of invalid input
  - Attempts to find vulnerabilities by exceeding the possible combinations of malformed input beyond the boundaries of the original client

- Best suited for:
  - Services using documented protocols HTTP, FTP, RPC, DCOM, …
  - Web applications
  - Protocols with many field combinations
Fuzzing

- Semi-Manual fuzzing
  - Writing scripts or short programs acting as rough clients
  - Manually changing the code for each test
  - Running the code and evaluating the response

- Automated fuzzing
  - Writing scripts or programs to iterate through a high number of invalid input
  - Running the code and letting it iterate until the target crashes
Fuzzing

- What you try to determine
  - Semi-Manual fuzzing
    - Unexpected responses
    - Modified data in the response
    - Changed timing behavior
    - Target crashes
  - Automated fuzzing
    - Target crashes
Fuzzing

- Semi-Manual fuzzing procedure
  - Get your script to work normally
  - Change fields one at the time
  - Generate output (send data, create file, ...)
  - Inspect results
  - Change fields again, depending on results
  - Generate output
  - Repeat last two steps
Example:
Symantec PC AnyWhere 10.5

- Timing issue with frequent reconnects and initial handshake
- Fails to synchronize load and unload of a DLL for the tray bar icon
- DoS: connect, handshake and disconnect about 10 times
Automated fuzzing procedure

- Define what vulnerabilities you want to look for
- Create iterator script/program using a fuzzer framework
  - Output data for every vulnerability type you want to test
  - Output data for multiple/combined vulnerabilities
  - Iterate through all combinations
- Wait until your target crashes
  - Needs a debugger attached to the target in case the vulnerability is hidden by a SEH handling it
  - Issues with „forking“ processes under Win32
Fuzzing Frameworks

- SPIKE
  - By Dave Aitel, Immunity Inc
  - Currently version 2.9
  - Block based fuzzer
  - Written in C
  - Fuzzing programs need to be in C too
  - Rudimentary functions for sending and receiving data, strings and iterations
  - Almost no documentation
  - Comes with a number of demo fuzzing programs
Fuzzing Frameworks

- **Peach**
  - By Michael Eddington, IOActive
  - Currently pre-release state
  - Written in Python (object oriented)
  - Consists of:
    - **Generators** for static elements or protocol messages
    - **Transformers** for all kinds of en/decoding
    - **Protocols** for managing state over multiple messages
    - **Publishers** for data output to files, protocols, etc.
    - **Groups** for incrementing and changing Generators
    - **Scripts** for abstraction of the per-packet operations
  - Documented fully, including examples
Fuzzing

- **Advantages:**
  - Semi-Manual fuzzing
    - „Try-Inspect“ Process leads to fast findings
    - Fuzzing script can be promoted to exploit
  - Automated fuzzing
    - Quickly uncovers a wide range of overflow and format string vulnerabilities
    - Effective when many combinations are possible

- **Disadvantages:**
  - Understanding of the underlying protocol required
  - Tester has to rely on fuzzer
  - Debugger on the target system often required
  - Can hide a bug behind another bug
Static analysis

- Disassembly of the target binary in order to find vulnerabilities.
- Identification of vulnerable code sequences independent of their location.
- Often paired with automatic analysis of calls to known library functions with vulnerability potential.
Static analysis

- Always a manual procedure with aid of several tools
- Requirements:
  - Binaries of the target
  - Interactive Disassembler (IDA)
  - Library reference for the target
  - Fluent assembly
  - Fluent C (!)
Static analysis

- Find *REACHABLE* references to functions with vulnerability potential: strcpy(), sprintf(), ...
  (Testifies to the average quality of COTS 😊)
- Check the call arguments for each reference if they suggest a vulnerability
  ```c
  sprintf( buffer, "\%s\", ...
  ```
- Check if the data can be influenced
  ```c
  sprintf( buffer, "\%s\", user_input );
  ```
- Find potential limiting factors
  ```c
  sprintf( buffer, "\%s\",
            strlen(user_input)>=(sizeof(buffer)-1) mimic"user_input);
  ```
Static analysis

- Reverse engineering of lower level protocol handlers
  - Find calls to recv(), recvfrom(), WSArecv(), WSArecvfrom(), read(), ...
  - Determine the buffer holding the data
  - Follow the program flow to eventually find the parsing functions
  - Reverse engineer the parsing functions
  - Identify potential for parsing mistakes
Static analysis

- Things to remember:
  - In order to find bugs, you should know the language better than the programmer: The ANSI C standard should be your Bible. Sleep on it. Read it. Worship it.
  - Knowing „not a whole lot of“ assembly will make you miss „a whole lot“ of bugs.
  - Reading diff‘s to open source software will make you a better closed source auditor, too.
  - Different compilers react differently, and an astonishing number of them do not implement the ANSI standard correctly.
  - The standard itself is often ambiguous ➔ things break.
C trivia question

Signedness of comparison is not always obvious:

```c
int a = 10, c = -1;
unsigned int b = 10;

if( a + b > c )
```

The above condition evaluates to FALSE

```c
int a = 10, c = -1;
unsigned short b = 10;

if( a + b > c )
```

The above condition evaluates to TRUE – Why?
Quoting from the ANSI/ISO C Standard, Page 57:

If an int can represent all values of the original type, the value is converted to an int, otherwise, it is converted to an unsigned int. These are called Integer promotions.

unsigned int b can not fit all values into a regular int, so a+b ends up being unsigned int. On the other hand, unsigned short b can easily fit all values into a regular int, so a+b ends up being signed.
Static analysis

- **Advantages:**
  - Finds vulnerabilities in code normally not executed
  - Quickly uncovers most format string vulnerabilities
  - Advanced vulnerability identification

- **Disadvantages:**
  - Needs lots of time, experience and skill
  - Disassembly is almost never complete
    - Library call identification fails
    - C++ and Delphi code hard to read
    - Packed or obfuscated code hard to handle
    - Not usable for ugly stuff (Visual Basic)
Diffing

- Identification of a vulnerability after it has been found and fixed.
  - The goal here is to identify the fix, in order to find the vulnerability.

- Reasons:
  - Most people don’t patch quickly enough
  - Diffing takes less time than auditing
  - Vendors sometimes fuck up the patch, giving out vulns for free
  - Vendors sometimes tell the public about bugclasses, so they give out vulns for free
  - No need to audit to break into a system if you can wait for the next update to come out
Diffing

- In patches, one needs to first find out what files are modified
- Filemon from Sysinternals
- Killing the update after the unpacking procedure but before the copy
- Static analysis of the patch program itself
Differencing

- Comparing two versions of a binary by hand takes very long
  - Find functions that are at the same address
  - Compare the number of functions
  - Compare the size of functions
- Automated binary differencing is far superior
  - Graph based fingerprinting of functions
  - Automated comparison
  - Can also be used to port function names
  
Runtime analysis

- Running the target in a debugging environment and inspecting the code during execution.
- Identification of vulnerable code sequences using disassembly, much like static analysis.
- Observation of the target code rather than completely reverse engineering it.
Runtime analysis

- Manual process with the aid of debugging tools

- Requirements:
  - Functioning version of the target
  - Debugger
  - Fluent assembly
  - Library reference for the target system
Release!
Phenoelit (dum(b)ug) core

- Complete and fully open source Win32 debugger core
- C++ class architecture
- PE parsing, disassembly, thread handling, breakpoints
- Instant debugger creation using a few lines of code

http://www.phenoelit.de/dumbbug/
Runtime analysis

- Data follow procedure
  - Identify functions that produce „incoming“ data, such as `recv()` and break there
  - Follow the data through the program flow to identify parsing functions
  - Following the data can be supported by memory breakpoints
  - Reverse engineer the parsing function, looking for mistakes in the programming
  - Craft data to trigger the suspected vulnerability and inspect the results
Runtime analysis

- Code follow procedure
  - Identify functions with vulnerability potential
  - Break every time such a function is executed and inspect the arguments
  - Check the arguments if they suggest a vulnerability in this case
  - Check the arguments if they are user supplied data or derived from it

- For most functions, this is impractical because of the high number of calls to them
  - Often, only one in 100 calls is relevant
Runtime analysis

- **Advantages:**
  - Correct disassembly at the time of execution
  - Known state of registers
  - Advanced vulnerability identification
  - Slightly faster than static analysis, due to skipping of uninteresting code

- **Disadvantages:**
  - Time, skill and experience required
  - „Break-and-Inspect“ not good for library functions
  - Timeout issues
  - Detaching of debugger only on Win2003
Static and Runtime analysis

- **Usual findings:**
  - Application and protocol level overflows
  - Format string vulnerabilities
  - Integer vulnerabilities

- **Best suited for:**
  - All kinds protocol parsers
  - Logging and data processing
  - Code using unsafe functions
Release!
Phenoelit (dum(b)ug) Ltrace

- Ltrace for Windows
- Log calls to any function
- Before and after states
- Call conventions
- Follows „forks“
- Stack analysis
- Format string analysis

http://www.phenoelit.de/dumbbug/
Trace defs

- Trace definitions used to identify arguments of traced functions
- Native C notation
- Argument directions
- Return value or output buffer matching

```c
int __cdecl recv(
    [in] int socket, [both] char * buf,
    [in] int len, [in] int flags);

"haxor" == int sprintf(
    [out] char * buf,
    [in] fmtchar * format );
```
Function call tracing

- Advantages:
  - Extremely fast
  - No disassembly
  - Recognition of user supplied data
  - Automagic format string vulnerability detection

- Disadvantages:
  - Incomplete: only called functions traced
  - Covers only unsafe functions
  - Does not (yet) identify compiled in incarnations of library functions
Combining forces

- Fuzzing and tracing
  - Fuzzing using Peach and a well designed script
  - Attaching ltrace or (dum(b)ug) tracer to catch exceptions and “forking”
- Semi-Manual fuzzing and static analysis scripts
Writing Exploits
Don’t write exploits in C
- Error prone
- Not portable
- Ugly

Connecting and sending data can be done in script languages
- Perl
- Python
- Even Java *urgl*
Exploit creation process

- Find a vulnerability
- Trigger it
- Get code execution
- Get shellcode there
- Execute shellcode
Trigger

- Make sure you can trigger the vulnerability on more than one computer
- Observe environment requirements

Example:
- Overflow in HTTP server
- Crash: GET /AAAAA.... HTTP/1.0\r\n\r\n
- Concatenation of real path and requested URI: c:\the\buggy\server\AAAAAAAA...
Code execution

- Influencing the instruction pointer (e.g. EIP) to point at data you influence
  - Use illegal instruction codes (0xFFFF, 0xC0) or 0xCC (debug break)
  - Don’t limit yourself to the overflowed buffer
- Try to get out of the function as soon as possible
  - Reading of local variables between overflow and return
  - Writing or local variables or function arguments
Code execution

- Function entry
- Return
- Use of variable BLA
- Buffer overflow
- Return address
  - AAAA
  - AAAA
  - AAAA
  - AAAA
  - AAAA
  - AAAA
Code execution: Return address

- Direct return into buffer
  - Mostly not reliable
  - Not a good idea with threads
- Return to known code location
  - JMP <Reg>
  - CALL <Reg>
  - POP <Reg>
  - POP <Reg> ...
  - RET
- ADD ESP,0x??
  - RET
- Structured Exception Handler
SEH in full color

1. Overflow up to SEH address
2. Trigger exception (not hard)
3. Get code exec from ntdll.dll
4. Enjoy

ntdll.dll

RetAddr
Useful JMP/CALL <reg> sequences in wide char addressable locations are very hard to find.

Solution: pure simple brute force

- Search the entire mapped address space for wide char addressable locations
- Search from those locations...
  - Bail if memory access occurs
  - Print result if JMP/CALL <reg> is found
  - Recourse if CALL/JMP <imm> is found

→ Find all addressable JMP/CALL <reg>
... while at it ...

... put an end to those return address issues
- Also support search for JMP/CALL <reg> in ASCII overflows
- Support automatic handling of forbidden characters such as 0x00
- Support stack-return as well
  - If a pointer to your buffer is further up in stack, adjust stack by n bytes and return
- Support saving the return addresses
- Support diffing of return addresses

→ Phenoelit OllyUni Plugin for OllyDbg
OllyUni finding example

- UNICODE return addresses that are not directly reachable:

0x00420153 is addressable by the sequence 0x429C in the ANSI table

0x0042015D is not Unicode addressable, but contains CALL EBX
Getting shellcode there

- Shellcode is a general term describing data that became code
- Shellcode creation process:
  - Write a small program that opens a listener with a shell
  - Compile
  - Disassemble
  - Make it position independent
  - Remove forbidden characters
Getting shellcode there

- When writing your own shellcode...
  - Try to make it flexible
  - Do not rely on libraries
  - Use delta offsets for variable addressing
- Use MosDef
- Use Metasploit
- Use Google

```assembly
pop ebp
...
```

```
EB 20
5D
...
E8 DB FF FF FF
2F 62 69 6E 2F 73 68
```
Getting shellcode there

- Shellcode can be transported in many ways
  - In overflowed buffer
  - (local) in environment variable
  - In other buffer of the same request
  - In data you sent before (username?)

- Shellcode often gets transformed
  - 0x00 characters in string operations
  - Forbidden characters
  - Encoding (eg. UNICODE transformation)
Unicode Shellcode

- Transformation example: UNICODE

```
E8 00000000 CALL 004015C5
5D           POP EBP
64:8B0D 000000 MOV ECX,DWORD PTR FS:[0]
```

```
E8 00000000 CALL 004015C5
0000 ADD BYTE PTR DS:[EAX],AL
0000 ADD BYTE PTR DS:[EAX],AL
005D 00 ADD BYTE PTR SS:[EBP],BL
64:008B 000D00 ADD BYTE PTR FS:[EBX+D00],CL
0000 ADD BYTE PTR DS:[EAX],AL
0000 ADD BYTE PTR DS:[EAX],AL
0000 ADD BYTE PTR DS:[EAX],AL
```
Venetian shellcode

- First published as "Creating Arbitrary Shellcode In Unicode Expanded Strings" by Chris Anley (chris@nextgenss.com)
- Chris dubbed the method "venetian shell code" due to the fact that the 0x00 gaps are closed like a venetian blind
- Implemented as shell code generator
  - Dave Aitel’s makeunicode2.py
  - Phenoelit’s vense.pl
Venetian code in color

1. Set one register to the start of your real shell code
2. Pad 3 bytes
3. Modify the 0x00 byte
4. Pad 3 bytes
5. Increase your pointer register
6. Goto 2

UNICODE
venetian code

| 01 | 02 | 03 | 04 |
| 05 | 06 | 07 | 08 |
Finalizing the exploit

- Now you can:
  - Trigger the vulnerability
  - Point the instruction pointer to your data (which becomes code, did you pay attention?)
  - Get your code (data 😊) there

- Try it
  - With debugger
  - Without debugger
  - Different OS versions
  - Different OS languages
Exploiting non-standard stuff
What is the difference?

- Other platforms are von Neumann machines as well
  - They got a CPU
  - They got Random Access Memory
  - They got (some) permanent storage
  - They got network connectivity

- Why not hack ...  
  - Routers
  - Printers
  - Cell Phones
  - Hotel Pay-TV sets
Non-Intel CPUs

- Differences from Intel to non-Intel
  - Often fixed size instructions (32bit)
  - Nicer instruction set
  - Alignment requirements
  - Different condition codes and conditional execution
  - Delayed branches
  - More registers
Exploiting other OSes

- Get a shell
- Get a debugger
- Understand the platform
  - Process management
  - Memory management
  - Privilege mechanism and granularity
- Read, but don’t believe
Exploiting black boxes

- Look for undocumented debugging or logging capabilities
  - Hidden functionality in shell
  - Not soldered serial port
  - JTAG interface
- Trail-and-Error will do as well
- Use illegal instructions or infinite loops as „stop“ of your code
- Interpret the results
Summary

- Read
- Think
- Do
- Have fun
Halvar‘s greetz:
  Too many to list here

FX‘s greetz:
  Phenoelit@home, TESO, THC, ADM, Gettohackers, all@ph-neutral, Shmoo, DEFCON goons, EEye, Rocky, HD Moore, cmn, Gera, Rocketgrl, a-few@CCC, c-base