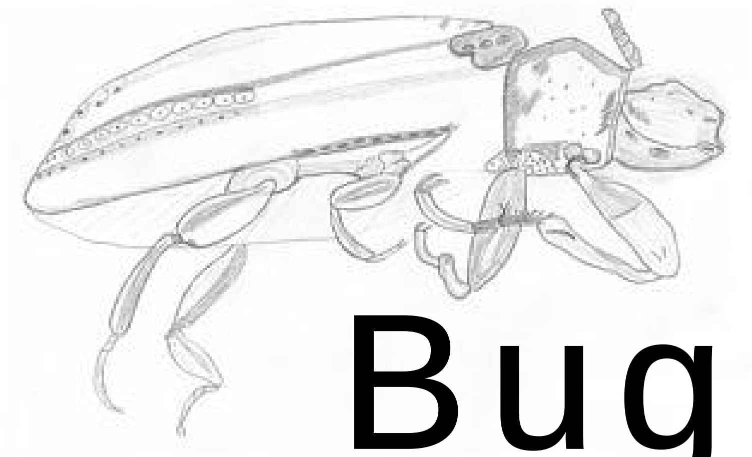
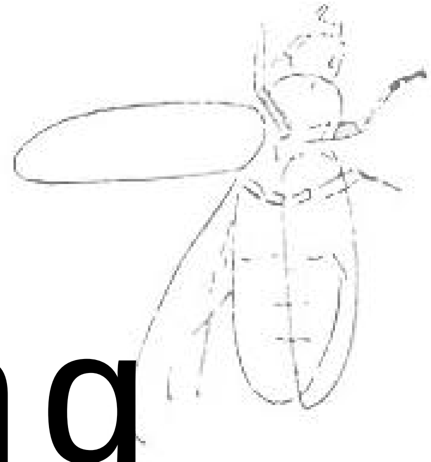




Bug hunting

Vulnerability finding methods in
Windows 32 environments compared

FX of Phenoelit

The goal: 0day

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

Phenoelit

The environment

- Windows NT / 2k / ((2k++)++)++
- Closed source binaries
- NT services
  - Often large binaries
  - Some times „forking“
- Application frameworks
  - IIS ISAPIs
  - Large scale frameworks (eg. SAP)
- Widely used clients

Obstacles

- Extremely large Win32 API
- Large, dynamically linked binaries
- Compiler specifics and optimization
- Use of library functions in-code
- Function inlining
- Vendor specific libraries replacing standard calls
- Unknown protocols
- Vendor specific obscurities

Phenoelit

Testing methods

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

Phenoelit

Manual Testing

- Using the standard client (or other counterpart) to access the target service
- Observing the behavior:
  - Valid input
  - Invalid input
  - Timing
  - Network communication
  - Pre-authentication handshakes
  - Common configuration tasks and failures
  - Target administration specifics

Phenoelit

Manual Testing [2]

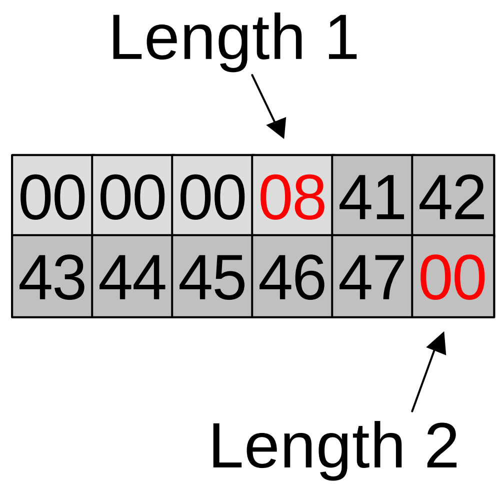
- What you try to determine:
  - States in the target
  - Reaction to valid input
  - Reaction to invalid input
  - Reaction to changes in timing
  - Information transmitted before and after authentication
  - Runtime environment requirements of the target
  - Default configuration and misconfiguration issues
  - Logging capabilities

Manual Testing [3]

- 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 [4]

- 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



Phenoelit

Manual Testing [5] - Pros

- No need for additional tools
- Becoming familiar with the target
- Un-intrusive
- Uncovers client side security quickly
- Easy correlation between user action and network traffic
- Takes configuration into account
- High abstraction level, no need to understand all the internals of the target

Manual Testing [6] - Cons

- Slow
- Potentially high learning effort
- Incomplete coverage – only functionality configured and used is tested
- 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 [7]

- 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
- Observing the behavior:
  - Not as closely as with manual testing
  - Responses are some times inspected
  - Often only crashes are considered

Fuzzing [2]

- 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 itterate through a high number of invalid input
  - Running the code and letting it itterate until the target crashes

Fuzzing [3]

- 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 [4]

- 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

Phenoelit

Fuzzing [5]

- 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

Phenoelit

Fuzzing [6] - 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 [7] - 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 absctraction of the per-packet operations
  - Documented fully, including examples

Fuzzing [8] - Pros

- Semi-Manual fuzzing
  - „Try-Inspect“ Process leads to fast findings
  - Same advantages as manual testing
  - Ability to prove the vulnerability
  - Fuzzing script can be promoted to exploit
- Automated fuzzing
  - Process guarantees known level of coverage
  - Quickly uncovers a wide range of overflow and format string vulnerabilities
  - Effective when many combinations are possible
  - Code reuse for known protocols

Fuzzing [9] - Cons

- Understanding of the underlying protocol required
- Incomplete coverage –
  only functionality configured and used is tested
- Automated Fuzzing
  - Test scripts need to be developed
  - Test scripts need to take potential target specifics into account
  - Tester has to rely on fuzzer
  - Debugger on the target system often required
  - Can hide a bug behind another bug

Phenoelit

Fuzzing [10]

- Usual findings:
  - Application level overflows
  - Format string vulnerabilities
  - Path traversal
- Best suited for:
  - Services using documented protocols
  - Standard servers: Web, FTP, LDAP, RPC, ...
  - Web applications (semi-manual fuzzing)
  - Protocols with many field combinations

Static analysis

- Disassembly of the target binary in order to find vulnerabilities.
- Identification of vulnerable code sequences independent of their location
- In some cases back-translation of the disassembly into a higher level language such as C.
- Often paired with automatic analysis of calls to known library functions with vulnerability potential

Phenoelit

Static analysis [2]

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

Static analysis [3]

- Identification of vulnerable code
  - Find references to functions with vulnerability potential: strcpy(), sprintf(), ...
  - Check the call arguments for each reference if they suggest a vulnerability

    ```
    sprintf( buffer, „%s“, ...
    ```
  - Check if the data can be influenced

    ```
    sprintf( buffer, „%s“, user_input );
    ```
  - Find potential limiting factors

    ```
    sprintf( buffer, „%s“,
      strlen(user_input)>(sizeof(buffer)-1)?“big“:user_input);
    ```

Phenoelit

Static analysis [4]

- Reverse engineering of lower level protocol handlers
  - Find calls to recv(), recvfrom(), WSArecv(), WSArecvfrom(), ...
  - 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

Automated Static analysis [5]

- Code flow analysis
  - Following branches and calls
  - Building a flow graph of the binary or subsections
  - Identifies functions, stack variables
  - Improves reverse engineering
- Automated library call identification
  - Finds calls to unsafe library functions
  - Output needs to be inspected by reverse engineer
  - Can automatically identify format string vulnerabilities

Static analysis [6] - Pros

- In depth analysis
- Finds vulnerabilities in code normally not executed
- Quickly uncovers most format string vulnerabilities
- Advanced vulnerability identification
  - Integer overflows and wraps
  - Off-by-one errors
  - Complex combined vulnerabilities
- Complete coverage of the code inspected

Static analysis [7] - Cons

- Extremely time consuming
- Experience and skill required
- Disassembly is almost never complete
  - Library call and inlinded function identification fails
  - Packed or protected binaries
  - Multiple level indirect calls to dynamic data (especially in C++ or Delphi code)
  - Code flow analysis fails
  - Structures and other advanced data structures hard to handle
- Not usable for higher level languages (Visual Basic)

Phenoelit

Static analysis [8]

- Usual findings:
  - Protocol level overflows
  - Complex vulnerabilities
  - Integer vulnerabilities
- Best suited for:
  - Protocol parsers
  - Unknown protocols
  - Code using unsafe functions
  - In depth analysis of critical code sections

Example:
Orenosv 0.6.0 HTTP server

- Combined logging buffer overflow
- Classic case of multiple sprintf() calls going wrong
- Remote
  NT Authority/SYSTEM

Phenoelit

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:
  - Vendors do not notify the public of an identified vulnerability but fix it silently.
  - Silent vendor fixes don‘t guarantee security, since the fix itself could be flawed.
  - For various reasons, some still want an exploit for vulnerabilities that are already fixed

Diffing [2]

- In patches, one needs to first find out what files are modified
  - Single file patches are easily identified
  - Higher number file replacements like in Microsoft hotfixes and Service Packs need to be monitored.
- Filemon from Sysinternals
- Killing the update after the unpacking procedure but before the copy
- Static analysis of the patch

Diffing [3]

- 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 diffing is far superior
  - Graph based fingerprinting of functions
  - Automated comparsion
  - Can also be used to port function names
  - Check http://www.sabre-security.com/ for magic

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 [2]

- Manual process with the aid of debugging tools
- Requirements:
  - *Functioning* version of the target
  - Debugger
  - Fluent assembly
  - Library reference for the target system

Phenoelit (dum(b)ug) core

(dumB)
(Bug)

dumb people (write)
dumb debuggers (to find)
dumb bugs

- 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/

Phenoelit

Runtime analysis [3]

- 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 [4]

- 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

Phenoelit

Runtime analysis [5] - Pros

- Just in time disassembly
  - Correct information at the time of execution
  - Known state of registers
- Quickly uncovers format string vulnerabilities
- Advanced vulnerability identification
  - Integer overflows and wraps
  - Off-by-one errors
- Slightly faster than static analysis, due to skipping of uninteresting code
- Exception catching

Phenoelit

Runtime analysis [6] - Cons

- Time consuming
- Skill and experience still required
- Break-and-Inspect not well suited for frequently called functions
- Requirements (CPU power, binaries, etc)
- Timing issues
  - Connection timeout during code inspection
  - Other timing related stuff breaking
- Detaching of a debugger only in Win2003

Runtime analysis [7]

- Usual findings:
  - Application level overflows
  - Complex vulnerabilities
  - Integer vulnerabilities
- Best suited for:
  - All kinds protocol parsers
  - Logging and data processing
  - Code using unsafe functions

Phenoelit (dum(b)ug) ltrace

(dumB)
(Bug)

dumb people (write)
dumb debuggers (to find)
dumb bugs

- 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/

Phenoelit

Trace defs

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

```
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

- Pros:
  - Extremely fast
  - No disassembly
  - Recognition of user supplied data
  - Automagic format string vulnerability detection
- Cons:
  - Incomplete: only called functions traced
  - Covers only unsafe functions
  - Does not (yet) identify compiled in incarnations of library functions

Example:
Orenosv 0.6.0 FTP server

- Logging buffer overflow
- sprintf(buffer,“%02X“,...) calls in a loop going wrong
- Very hard to identify with static analysis
  - Ignore previous statement if using Halvar‘s tools
- Remote
  NT Authority/SYSTEM

Phenoelit

Combining forces

- Team 1
  - Fuzzing using Peach and a well designed script
  - Attaching a debugger to catch exceptions
- Team 2
  - Call trace using (dum(b)ug) tracer
  - Manual testing using existing client
  - Script for sending suspected overflow data
- Team 3
  - Disassembly using IDA
  - Semi-Manual fuzzing according to disassembly

Phenoelit

Summary

- Different vulnerability testing methods are available, each with different properties and areas of application.
- For quick bug finding, automated methods are best.
- For thorough analysis, manual methods and static analysis should be preferred.

Phenoelit

GREETS

Gera

Ghettohackers

Halvar

cmn

Stealth

Jeff & Ping

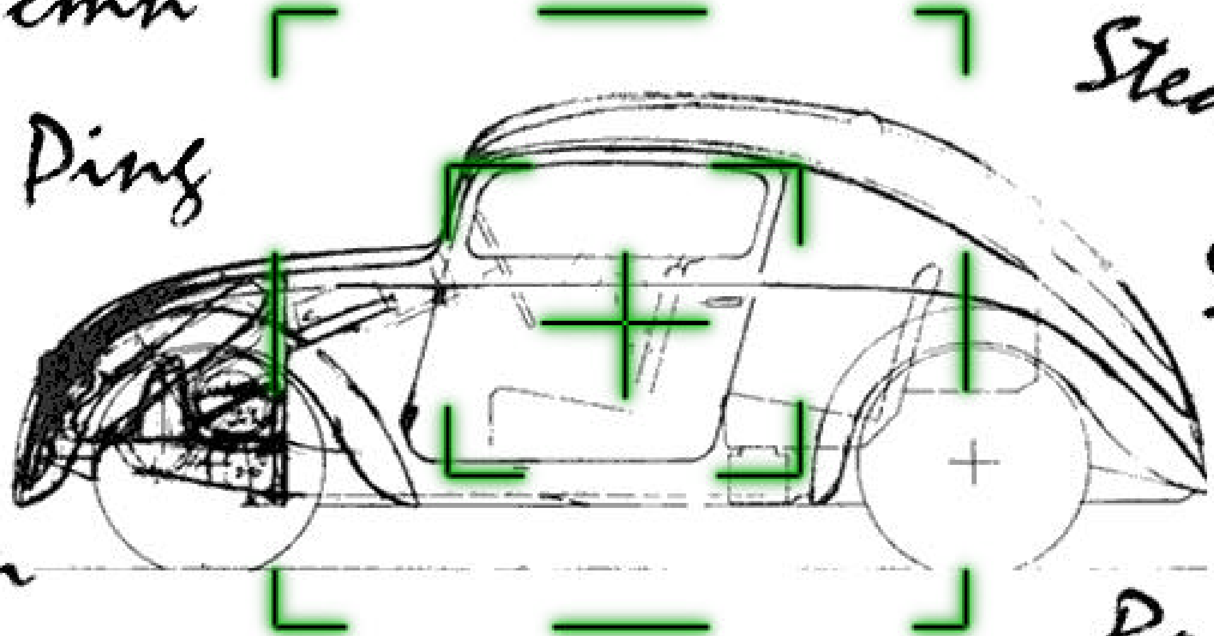
Scut

dd

EEye

Spoonm

Rocketgrl



Bug detected... analyzing...
[data] adding new bug class ' VW '
Exploiting...

HD Moore

Scyper & Gamma

Shmoo

Phenoelit