Runtime Decompilation

The ‘GreyBox’ process for Exploiting Software

By: Greg Hoglund
hoglund@hbgary.com
Introduction

- This presentation is an overview of the lab process used by HBGary™ to locate and exploit software bugs.
- The value is reduced time investment and formalizing a ‘black art’.
The Deconstructionist

- Taking a system apart is about uncovering mysteries
- Having secret knowledge attracts the human psyche at the deepest level
- There is an age old battle between those that create systems and those who take them apart
  - i.e., cryptoanalysis
Why Exploit Software?

- Exploits are worth money
  - A vulnerability can be worth over $100K
    - The vendor costs are huge for a public vulnerability
  - An exploit costs less than physical 'bugs'
- Exploits are worth lives
  - An exploit is safer than physical penetration
- Exploits are strategic
  - Disable or control the information systems of your enemies decision cycle
Survivability

- Exploits have a lifetime
- Every use of an exploit has the potential to compromise the asset
- Exploits depend on bugs and your enemy may also find the same bug
- The public may find the same bug
- Once public, many exploits can be protected against or detected via an IDS
In order to maintain a battle advantage, your offensive information capability must include a lab process for finding and exploiting new software bugs.
Chapter One

The Bugs!
The Bugs

1. Buffer Overflows
   a. Lack of bounds checking
   b. Arithmetic errors
2. Parsing Problems
   a. Input filters and normalization
3. General State Corruption
4. Race Conditions
Buffer Overflows

- Old News, but still most common today
  - Because of speed, most server software is still developed in c/c++

- Will remain common until old compiler technologies are abandoned
  - Strongly typed languages, such as C#, eliminate simple string overflows
Parsing Problems

- Not solved by better compilers
- Solved only by good algorithms
- To eliminate parsing problems requires standardized algorithms
  - Similar to peer review on crypto systems
- This will never happen
  - Parsing problems here to stay
General State Exploits

- States control decisions
- Users can cause state transitions
- Some states are insecure by nature
- State exploits are found by sending commands in the exact order and context to arrive at the insecure state
- Only solved by provably correct systems
  - Humans are never going to build provably correct systems
Race Conditions

- State problems are going to be difficult to measure and control.
- When state is managed over many nodes, the problem becomes even harder.
  - When state must be synchronized among nodes, we have race conditions.
  - The problem is compounded greatly.
  - This is the ‘buffer overflow’ of the future.
Chapter Two

What is the ‘GreyBox’ process?
GreyBox:
Combining both static analysis and runtime fault injection to maximize coverage of a software programs’ state-space. Typically used to detect and isolate fault states in a software system.
White Box

- In theory, operating with full knowledge about the inner workings of the system
  - At best, we only have an approximate understanding of the builder’s intent
- White box analysis involves “deadlistings” static disassemblies of the binary
  - Source code is an added advantage, like having really good documentation for the deadlisting
- The software is *not* being executed
First Pass

- When confronted with a new binary, the HBGary team fires the binary through BugScan™
- We obtain a report within minutes to assess whether the programmers use secure coding practices
  - Typically they do not
- We use the BugScan report to prioritize which binaries will be analyzed first
  - Binaries with bad reports are hit first
Manage the Deadlisting

- IDA-Pro allows you to manage and comment a large deadlisting
Is it actually exploitable?

- Depends on many variables in the environment
- *All* automatic analysis tools have this problem
- It almost *always takes an expert* reverse engineer to determine if a condition is exploitable
Does it matter?

- Even if a vulnerability cannot be reached today – what can you say about tomorrow?
- What if interface changes?
- What if code gets used from other locations?
- Is the original author going to be maintaining this code in 10 years?
Automatic Bug Detection

- The bug must have a defined pattern, it must be *schematic* in nature.
- Effective when certain conditions exist:
  - Availability of type information
  - Separation of data and code
  - All instructions can be recovered
  - Data that drives control flow can be mapped
Branching Decisions

- Many branches are made based on values that are calculated at runtime.
- The static analyzer must emulate execution to determine these values.
- At some point, the emulation becomes computationally equivalent to running the program in the first place. How much emulation is enough?
Backtraces reach dead ends

○ We backtrace up to 64 steps from a vulnerable function
  ● Every branch is exercised

○ Back traced cross references can be used to connect input with a code location
  ● For example, does a previous function take input from the network?

○ Many times a static backtrace dead-ends
  ● Windows message handler
Black Box

- All we see are the outputs from the software – no inner workings

- Requires deep protocol knowledge

- ‘Fuzzers’: 
  - Hailstorm and Spike
Black Box is not stand alone

- Black box testers take FOREVER to complete their input sequences.
- If the program is slow, this compounds the problem
- Amounts to ‘brute forcing’
- Finding bugs with pure brute force is mostly luck
Blackbox State

- Typical network software is highly stateful.
- A client must be able to maintain a complex state in order to communicate effectively with the target.
- Modeling highly stateful clients from scratch is very time-consuming and prone to error.
Instrumented Clients

- Using a real client program eliminates most of the state issues
- You don’t need to rebuild the wheel
- Fault injection is inserted *in-transit* by modifying the code within the client program
- The client program becomes a *hostile mutant*
Fault Injection Clients

- If the protocol is proprietary you have two choices
  - Modify in the middle the packets
    - Only works is protocol is not overly complex and not encrypted – beware of authentication/encryption
  - Instrument proprietary client
    - Requires difficult call-hooking, time consuming
Hooking clients

- Find location where pointer is held in register
- Put breakpoint on this location and modify the given string in memory
  - Cannot BO string w/o corruption
- Or, replace the pointer with another pointer
  - May cause state problems in some clients
- Using ‘debugger’ technology makes this whole process easier – no EXE patching required
GreyBox

- Combine black-box injection with code analysis
- If you use a program debugger, your performing grey-box analysis
- Performed at runtime so software can be observed
- All instructions which are executed can be obtained. All data involved at these points can be tracked
Interactive

- Grey-box testing is an interactive process between a skilled engineer and the target program.
- Tools used include SoftIce, OllyDbg, Aegir, Fenris, GDB, Tempest, and the MS-Visual C debugger*.

*IDA-Pro has an integrated debugger, but the current version is not evolved enough for industrial level work.
Fenris

www.bindview.com
Chapter Three

How to use Bug Scan™
Easy Stuff – Introducing BugScan!

- BugScan is **extremely simple** to use
- Submit binary and get report
- Report cannot verify if conditions are actually exploitable
  - But it takes 30 seconds, not 30 hours
  - Defensive stance – don’t wait for someone to attack before you protect yourself
Submit a File

BugScan v1.0  Browse Reports  Analyze a File  Log Off

Select a file to analyze:

Browse...

Submit

File uploaded successfully to the Web server
View the Report

![BugScan: Report - Microsoft Internet Explorer](image)

### BugScan v1.0

<table>
<thead>
<tr>
<th>ID</th>
<th>Sev</th>
<th>Name</th>
<th>Risk</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>wsprintfA</td>
<td>overflow</td>
<td>Replace this call with the more secure call, <em>wscanf</em>. <code>wsprintf</code> is a variant of <code>wscanf</code> where the user explicitly specifies the length of the destination buffer. This feature helps avoid the possibility of the destination buffer being written past. Though not officially a part of the ISO C99 standard, this call is available in most modern compilers. Two problems can still persist: format string bugs and specifying an incorrect length for the destination buffer.</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>_strncpyA</td>
<td>low</td>
<td>Double check that the destination buffer is larger than the specified length in this call. If the supplied string is exactly equal to the size of the destination buffer, the string will not be NULL terminated.</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>_strncpyA</td>
<td>overflow</td>
<td>Replace this call with the more secure call, <code>_strncpy</code>. <code>_strncpy</code> is a variant of <code>_strncpy</code> where the user explicitly specifies the length of the destination buffer. This feature helps avoid the possibility of the destination buffer being written past. Though not officially a part of the ISO C99 standard, this call is available in most modern compilers. Two problems can still persist: format string bugs and specifying an incorrect length for the destination buffer.</td>
</tr>
</tbody>
</table>

*Discussions not available on [http://216.27.185.83/](http://216.27.185.83/)*
Latest BugScan Reports from the Field

TO BE REVEALED AT CONFERENCE
FREE BUGSCAN for BLACKHAT

- Use this logon to scan any binary, free for blackhat attendees for the next 60 days

  - HTTP://www.hbgary.com/freeblackhat
Chapter Four

How HBGary Uses GreyBox
Hard Stuff

- Designed for experts
  - Not a product!
- Requires reverse engineering skills not limited to:
  - Runtime debugging
  - Assembly code
  - Protocols
  - Technical knowledge of programming bugs
Introducing TEMPEST

Free technology available for download from www.hbgary.com
TEMPEST

- Connect the inputs with the bugs
- Verify the exploit
- Build a working exploit
- Offensive stance – find working injection vectors
- Defines a WORKFLOW
Static backtrace from suspect locations

Blocks of code that lead to sprintf

sprintf call
‘Self Learning Coverage’

- We start with user-supplied data
- We can detect when decisions are calculated from user input
- We can freeze and restore the program at any point and test new values
- Thus, we can map how user-controlled values influence state transitions
Location Coverage

- As program is used, if a code block is visited it will be highlighted ‘grey’*
VirtualQueryEx

MEM_COMMIT
&&
NOT PAGE_READONLY
&&
NOT PAGE_EXECUTE_READ
&&
NOT PAGE_GUARD
&&
NOT PAGE_NOACCESS
Fly-By’s & Drill Downs

- If we hit code blocks ‘above’ a suspect location we are alerted to potential operations that will cause the target to be exercised.
- Coverage helps us tune our input data to drill down to a target location.
  - This is the fundamental advantage.
Tracing

The code block exits to all these points.

Selected Code Block
Trillian IRC DLL

Signed/Unsigned mismatch in subroutine at 0x1000FE40
Boron Tagging

- Traces from known points
- Breakpoints on suspect calls
- Can be used as a strategy to skip large sections of the graph
  - These become ‘clusters’
  - We cannot create a spanning tree graph unless everything is connected
Leap Frogging

recv(...)

Change page protection in order to track access

mov edx, [esi]

lea edx, [esi]

mov [ecx], eax
Leapfrog with Boron

- Read memory to find all boron strings
- Set memory breakpoints on all these locations
- Locations are typically re-used
- Doesn’t always work because memory is cleared after use
Data Flow Analysis
Registers

Heap or stack

Write
Read
Graphing Problems

- Graph complexity increases with the number of back traces.
- Using tempest on more than a few target points at a time results in a huge, unwieldy graph.
Advanced Graphing

- Different graphing algorithms can be used
- Hyperbolic graphs serve better for browsing a large number of nodes
All code paths leading to sprintf call in commercial FTP server, information obtained statically
Filtering the set

- Don’t worry about sprintf if the format string doesn’t contain %s
- Don’t worry about off by ones if the size parameter is less than the stack correction
- Don’t worry about *anything* if the source data is not obtained from outside the function
ID locations using static analysis

Static traces

Backtrace from potentially vulnerable API call or location
ID locations using static analysis

Static traces

FUZZ

Locations which are visited are tagged grey
This is a HIT
- This causes a work item to be exercised.
Is user-supplied data used in the suspect call?

<table>
<thead>
<tr>
<th>Hits</th>
<th>Time: 12:25:57:257</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX: 08984058(144195672)</td>
<td>SELECT * FROM ACCOUN</td>
</tr>
<tr>
<td>EBX: 00B4F0F4(11858164)</td>
<td>.w.IL</td>
</tr>
<tr>
<td>ECX: 00000014(20)</td>
<td></td>
</tr>
<tr>
<td>EDX: 00000014(20)</td>
<td></td>
</tr>
<tr>
<td>ESI: 00B4F7AC(11859884)</td>
<td><a href="mailto:X@.k">X@.k</a></td>
</tr>
<tr>
<td>EDI: 0000002A(42)</td>
<td>.</td>
</tr>
<tr>
<td>EBP: 004A0604(4851204)</td>
<td>SELECT * FROM GROUPS</td>
</tr>
<tr>
<td>ESP: 00B4F0C0(11858112)</td>
<td>X@.</td>
</tr>
<tr>
<td>+0: 08984058(144195672)</td>
<td>SELECT * FROM ACCOUN</td>
</tr>
<tr>
<td>+4: 004A0604(4851204)</td>
<td>SELECT * FROM GROUPS</td>
</tr>
</tbody>
</table>
| +8: 00B4F0F4(11858164) | .w.
| +12: 77121644(1997674052) | .D$|
| +16: 003E4F50(4083536) | .5J |
ID locations using static analysis

Static traces

FUZZ

Faults?

%d but also handles a SELECT statement which looks mildly interesting.

0012f7e0 004a2070 ASCII "UPDATE SERVERSETTINGS SET SITE_SERVER_STATE=%%s WHERE ID=1"
Faults?

Unresolved Branches?

Use Data Flow Analysis to determine if branch is calculated from user-controlled data

Incomplete branch coverage

Modify input fuzzer to compensate
This location is the nearest fly-by. To solve the problem we must visit this location and determine what data is being used to make the branching decision.

In most cases, the value is not directly controlled by the fuzzer. This means that we must trace back further to determine if the value is calculated from user input. This is both tedious and time consuming.

wsprintf that uses %s

** this graph generated from commercial proxy server (vendor not revealed)
Conclusion

- There exists a process to connect user-input to potential vulnerabilities
- By tracing data and control flow at runtime, a fuzzer can be tuned to target a location
- Only a certain percentage of those bugs identified statically will be exploitable
Closing Remarks

BugScan is a commercial product that can be obtained from www.hbgary.com

Spike is free and can be obtained from www.immunitysec.com
Closing Remarks

- The Tempest debugging system is used internally by HBGary and is not a commercial product.
- Many components of the Tempest system are open source and can be obtained for study.

www.hbgary.com
Thank You

Greg Hoglund
HBGary, LLC.

hoglund@hbgary.com