FINDING VULNERABILITIES THROUGH BINARY DIFFING

이승진 (LEE SEUNGJIN)
SECURITY CONSULTANT AT GRAYHASH
고려대학교 정보보호대학원 (KOREA UNIVERSITY)
BIO

• Lee SeungJin (aka beist) from Korea University
• Principle security consultant at GrayHash
• SECUINSIDE organizer
• Consulting for big companies
• 5 times DEFCON CTF quals
• Over 10 times CTF win and run
• Hunting bugs is my favorite hobby
About this talk

• 50% - Technical
• 50% - 0day and demo
• Hunting security bugs through binary diffing
• Analyzing an 0-day of popular online game module
• Exploiting and demonstrating
What is binary differencing

- Comparing a binary to another binary
- Getting results that say the differences
- Very useful especially for reversers and exploiters
  - And I hope to find it useful for 0-day hunters as well
Binary diffing tools

- Bindiff of Zynamics (Acquired by google)
- DarunGrim2
- Patchdiff
- Turbodiff
- EBDS
- IDACompare
Advantages of diffing

- Binary diffing is used for
  - Patch analysis (1 day)
  - Identifying symbols (static compiled)
  - Platform-independent difference analysis (ARM vs x86)
  - Malware detection (VxClass)
Advantages of diffing

- Binary diffing can be used for
  - License check (open source licenses issues)
  - Bug hunting (Via finding vulnerable functions)
License check

- Some big companies in Korea are having trouble in open source licenses issues
- Developers often make mistakes that they don’t realize they are using open source licenses
- Would be very bad if they don’t mention it
  - I’ve heard they need a solution for this
License check

- Because the license check team can’t access code, they need a solution on binary-level
- Binary diffing can be a solution
  - Even it works well for different platforms like ARM vs x86
Finding vulnerable functions

• This idea popped up when I was preparing a lecture about diffing different arch-binaries
  • BinDiff was working really well for that
• And, I was thinking like
  • “Oh, it even says Similarity, what about we make a set of vulnerable functions and just compare it to binaries?”
Idea

• This idea relies on binary diffing
  • Especially, CFG (Control Flow Graph) based diffing
  • But, home-made sauce needed (And will be more)

• The idea is simple
  • It’s possible to make patterns of vulnerable functions
  • Then, comparing them with binaries and finding similar functions by binary diffing
Goal

- Better than "b dangerous_functions" like "b strcpy" or "b memcpy"
- Better than just some heuristic methods like loop-detection
- Pattern matching for both code level and binary level
Bug hunting on binary-level

- Two general approaches
  - Fuzzing
    - Can be extremely sophisticated (smart fuzzing)
    - Popular way for bug hunters these days
  - Dynamic + static analysis
    - Pure reversing or dynamic test
    - Still, static reversers find awesome bugs which fuzzers won’t find usually
Binary diffing for bug-hunting

- This idea could be a new approach for finding security bugs
- But it’s very naive yet
- Let’s tour technical stuff
More about diffing

- This talk is not about binary diffing itself
- But we should know some basic stuff
- Many of diffing tools are based on CFG
  - Also, instructions matching, basic blocks fingerprinting
BinDiff

- We chose BinDiff, because
  - Didn’t want to waste time to know how to use tools
  - It’s commercial which means they support customers
  - Most popular diffing program in the field
- BinDiff
  - From zynamics acquired by Google now
  - Only $200 USD (Reasonable price)
  - BinDiff can compare binaries supported by IDA
  - Using IDA to get CFG
How bindiff works

- BinDiff doesn’t care about the concrete assembly-level instructions
  - Except for generating hash on basic block level
- It’s more based on “Structural matching”
How bindiff works

- Structural matching
  - It makes fixedpoints between two binaries
  - Then tries to find more fixedpoints iteratively
  - When matching functions is done, it goes for basic blocks
How bindiff works

- Function matching strategy
  - hash matching (very good quality)
  - prime signature matching (SPP) (good quality)
  - string references (medium quality)
  - loop count matching (poor quality)
  - call sequence matching (ver poor quality)
  - etc

[According to zynamics manual]
How bindiff works

- Basic block matching strategy
  - hash matching (very good quality)
  - edges MD index and the other series (good quality)
  - loop entry matching (poor quality)
  - instruction count matching (very poor quality)
  - etc

[According to zynamics manual]
How to implement our idea

• It’s not easy to implement the idea
• Again, this project is still naive
  • But we’ll see progress at least
• We’ll explain step-by-step
Very basic example

• This code is vulnerable obviously

```c
void vuln_sample1(char *str) {
    char buf[256];
    strcpy(buf, str);
    printf("%s\n", buf);
}
```

• Let’s compare the code with this code

```c
void vuln_sample3(char *str) {
    char buf[256];
    printf("test\n");
    strcpy(buf, str);
    printf("%s\n", buf);
}
```
BinDiff result

• It seems BinDiff works well
• It says “0.69” similarity and “0.95” Confidence
  • Similarity is from 0 to 1
  • Confidence means which method it used to diff
BinDiff result
Another example (problem #1)

• What about this code

```c
void vuln_sample2(char *str) {
    char buf[256];
    char *p;
    printf("this is a vuln_sample2\n");
    strcpy(buf, str);
    p=strstr(buf, "beist\n");
    if(p)
        printf("beer\n");
    else
        printf("crying\n");
}
```

• BinDiff says “0.21” similarity and “0.38” confidence

• But we know it’s still vulnerable, this is the first huddle
What happens

- Disassembly of sample 1
What happens

- Disassembly of sample 2
- Branch/function calls are the problem
What happens

- Looks those graphs are different each other
- But if we see them as a view of a bug hunter, it’s the same flow semantically

```assembly
mov    eax, [ebp+Source]
push   eax             ; Source
lea    ecx, [ebp+Str]  
push   ecx             ; Dest
call   _strcpy
```
An idea to solve

- We try to remove instructions that don’t affect control flows

- In `vuln_sample2()`, we don’t need to count
  - `printf()`
  - `strstr()`
    - Should be counted in some situations
  - Like if the returned pointer is referenced for some string routines later
An idea to solve

• We only implemented a script that replace calls to NOP
  • But should be improved as below

• Implementation idea
  • Find non-interesting calls
  • Kill them
  • Adjust offsets

• Implement won’t be easy for complex routines
  • How to remove branches?
Complier dependency  (problem #2)

- What if it’s compiled by a different complier?
- Disassembly of sample 2 by GCC 4.6
Complier dependency (problem #2)

```
push    ebp
mov     ebp, esp
sub     esp, 144h
push    ebx
push    esi
push    edi
lea     edi, [ebp+var_144]
mov     ecx, 51h
mov     eax, 0CCCCC000h
rep stosd
push    offset aThisIsAVuln_sa ;"this is
call    _printf
add     esp, 4
mov     eax, [ebp+Source]
push    eax ; Source
lea     ecx, [ebp+Str]
push    ecx          ; Dest
call    _strcpy
add     esp, 8
push    offset SubStr   ;"beist\n"p
mov     eax, ecx
mov     esi, edi
mov     edi, edx
shr     ecx, 2
rep movsd
mov     ecx, eax
and     ecx, 3
rep movsb
lea     ecx, [esp+110h+Str]
push    ecx ; Str
call    strstr
```
An idea to solve

- Detect which compiler was used
  - Heuristic (Code generating style, etc)
  - Signature matching (raw bytes)
  - We used “EXEINFO PE”
- Optimization level could be known
  - Function prologue/epilogue analysis
  - Or we just can compile our subset for every optimization level
Same class bugs (problem #3)

```c
void vuln_sample1(char *str) {
    char buf[256];
    strcpy(buf, str);
    printf("%s\n", buf);
}
```

```c
void vuln_sample4(char *str) {
    char buf[256];
    memcpy(buf, str, strlen(str));
    printf("%s\n", buf);
}
```

- This can be solved somewhat easily
  - by grouping them
  - example) group copy_api = {strcpy, memcpy, gets, ++}

- IDAPython will be helpful
  - Changing call functions in force
  - But we have not been able to implement it yet
More, more, more problems

• Should we ignore operands?
• What about different architecture?
• How to group loop code and copy APIs together?
• etc
Categorize problems

• Most problems can be categorized into 2 groups
  • 1 - Different codebase: coding style, uninteresting functions, etc
  • 2 - Which compiler, version, optimization, etc
Patterns

- We’ve implemented around 20 patterns
  - dangerous APIs
  - loop
  - off-by-one
  - no null check
  - etc
- All of them are memory corruptions related
- Future work: patterns for other class bugs
  - use-after-free style would be hard
Case study

• We tried to find 0days from online game

• Why online game?
  • Because it has not been security-evaluated
  • Vendors don’t care user to user security
  • User to user hacking is much more dangerous
    • And more difficult than speed-hack hacking
  • Imagine that you get hacked because you just play online games
Case study

- Our target is a module by Kamuse company that does p2p file transmission
  - Users download install files from other users
  - This is reasonable since some games’ filesize > 10GB
- Almost NCSoft games use the module
- **What we got**: remote memory corruption bugs
Our strategy

- As this project is naive yet, we first try to find 0days by fuzzing
- Figure out of the bugs
- Compare the vulnerable functions with the binary
- To see if there are similar functions (or more bugs) that can be found via binary diffing
The kamuse module

- The module uses multiple ports
  - For p2p operation and transmission
- It does UPNP
  - So users can be reached by other users even if they are in home-routers
Fuzzing

- We set up a network fuzzer for the module
- Captured operation packets to get proper payload
- Manipulated the payload
- Got some 0days
  - Only took a look at one bug which is exploitable
Quick analysis

- Basic buffer overflow (No-bound-check)
  - But attackers have to have proper payload to reach the vulnerable function
- We can fully control EIP
  - Even we can use null-byte
- Let’s take a look at IDA
  - The vulnerable function is 0x00471d40
Diffing strategy

• Let’s say we have “target.exe” and sub_a() is a vulnerable function

• Looks BinDiff doesn’t show us more than 1 matching function

• If there are 90% and 70% matching functions, BinDiff only shows 90% function
Diffing strategy

• Copying just row bytes of \textit{sub\_a()} won’t work
  • Because all calls will be broken unless we adjust every offset - but would be not easy
  • If our calls are broken, diffing score would be very low
**Diffing strategy**

- **Idea**
  - copy target.exe (now we have target2.exe)
  - delete all functions of target.exe that start with “sub_”
    - except sub_a()
  - delete sub_a() of target2.exe
  - compare target.exe and target2.exe

- **Limitations**
  - What about if functions don’t start with “sub_”?
  - If we kill functions, cross references will be gone too
Diffing strategy

{target.exe}

{target2.exe}
Differing strategy

If almost same?
Result

- Found an interesting function
- Remember, 0x00471d40 is the vulnerable function
- 0x00471f60 is “0.77” similarity according to BinDiff
Result
Demo

- The kamuse module attack
  - Proof of concept code pops up an image if it works
Conclusion part 1

• Diffing for hunting bug is far away yet from the real world

• Many things to do
  • Very complex reversing works
  • Solving uninteresting instructions / branches problems
  • Compiler issues
  • Integration with code coverage, etc
Conclusion part 1

- We’re not saying we have to use BinDiff or specific approach to diff
  - Any way about diffing art would be helpful
- We’ve seen that it is possible
  - MS08-067 could have been revealed before ;)}
Conclusion part 2

• Many NCSOFT games use the module
  • When you download over 10GB files, you are very vulnerable
  • And, soooo many users
  • Scanning million ports in an hour is reality (Check out Immunity’s new service)
  • Thought “If a module is vulnerable which NCSOFT didn’t make, NCSOFT doesn’t have any responsibility?”

• Hint - google pays for webkit bugs.
• And.. there would be bugs in modules by NCSOFT too
Q & A

• Thanks to ISEC 2012!
• EMAIL: beist@grayhash.com
• BLOG: http://grayhash.com