Hacking Blind
BROP

Presented by:
Brooke Stinnett

Article written by:
Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazieres, Dan Boneh
The Question:

“Is it possible for attackers to extend their reach and create exploits for proprietary services when neither the source nor binary code is available?”
Objective: Hacking Blind

- write remote stack buffer overflow exploits without possessing a copy of the target binary or source code
- hack proprietary closed-binary services, or open-source servers manually compiled and installed from source where the binary remains unknown to the attacker
BROP: What is it?

- Blind ROP: remotely locates ROP gadgets
- Effective against server applications with a stack vulnerability and that restart after a crash
- BROP works against ASLR, NX memory, and stack canaries
- Targets 64-bit Linux, not Windows!
A Recap of ROP

- Developed to defeat defenses based on NX memory
- Links together short code snippets (gadgets)
- Used to gain control of programs without any dependence on code injection
- Each gadget ends with a return so the next gadget can execute
- The goal: to build an instruction sequence that spawns a shell
On to BROP...

- What we need:
  - a stack vulnerability and knowledge of how to trigger it
  - a server app that restarts after a crash
- The threat model:
  - an attacker that knows an input string that crashes a server, and can overwrite a variable length of bytes including a return instruction pointer
The Key Phases

- Stack reading
  - read the stack to leak canaries and a return address to defeat ASLR
- Blind ROP
  - find enough gadgets to invoke “write” and control its arguments
- Build the exploit
  - dump binary to find enough gadgets to build a shellcode
  - launch the final exploit
The BROP Gadget

- restores all callee saved registers
- misaligned parses of it yield a pop rdi and pop rsi
- by finding a single gadget, we find two gadgets!

Figure 7. The BROP gadget. If parsed at offset 0x7, it yields a pop rsi gadget, and at offset 0x9, it yields a pop rdi gadget. These two gadgets control the first two arguments to calls. By finding a single gadget (the BROP gadget) one actually finds two useful gadgets.
Finding Gadgets

- Scan the application’s text segment by overwriting the saved return address with an address pointing to text, and inspect the program’s behavior
- Either the program will crash (connection will close), or it will hang (connection stays open)
  - If the program doesn’t crash, you've found a gadget!
- Gadgets that stop program execution (stop gadgets) are fundamental to finding other gadgets
The Stop Gadget

- A stop gadget is anything that would cause the program to block
  - like an infinite loop or a blocking system call (like sleep)
- Acts as a signalizing mechanism
- Each time a useful gadget that doesn't cause a program crash is found, the stop gadget will run, blocking the program and leaving the socket open (instead of causing a crash)
- Then, you can scan the entire text segment to compile a list of gadgets
Identifying Gadgets

- **Probe**: the address of the gadget being scanned
- **Stop**: the address of the stop gadget that will not crash
- **Trap**: the address of non-executable memory that will cause a crash. “*It’s a trap!*”

By varying the position of the stop and trap on the stack, one can deduce the instructions being executed by the gadget, either because the trap or stop will execute, causing a crash or no crash respectively.
Finding a “call write”: The PLT

- Instead of finding two gadgets, we can find a single “call write” instruction
- We can find this conveniently in the Procedure Linking Table (PLT)
  - a jump table that contains all external library calls made by the application
- The PLT is the first region to contain valid executable code
Finding the PLT

- The PLT dereferences the GOT and jumps to the address stored in it.
- Most PLT entries won’t cause a crash (return EFAULT).
- If a couple addresses 16 bytes apart don’t cause a crash, and +6 bytes doesn’t crash, you’ve probably found the PLT!
Controlling rdx via strcmp

• We need to identify the different PLT entries
  • feed in different pairs of arguments and see what happens!
  • if there is no crash, you’ve found strcmp
• Then, you can set rdx to a non-zero value
• Now we have control over the first 3 arguments to any call: the first 2 via the BROP gadget, and the third indirectly via strcmp
Finding Write and Concluding the Attack

- At this point, we can find “write” by scanning each PLT entry and forcing a write to the socket and checking whether the write actually occurred.
- Then we can write the entire text segment to the attacker’s socket.
  - and disassemble it
  - and find more gadgets!
  - and dump the symbol table too!
Attack Summary

☐ Find where the executable is loaded

☐ Find a stop gadget and the PLT

☐ Find the BROP gadget to control the first two arguments to calls

☐ Find strcmp in the PLT; now you have control of the first three arguments

☐ Find write in the PLT; now you can dump the entire binary to find more gadgets

☐ Build a shellcode and exploit the server
BROP Prevention

- Rerandomization
  - rerandomize canaries and ASLR as often as possible
  - fork and exec the process on a crash or spawn
- Sleep on crash
  - delay a fork after a segmentation fault
  - can slow down an attack so an admin can come to the rescue
- ROP protections
  - Control Flow Integrity to enforce the control flow graph
- Compiler techniques
  - insert runtime bounds checks on buffers