CSE 509 Lecture 6

Prof. Rob Johnson, Scribe:Arun Rathakrishnan

September 18, 2013

1 Return Oriented Programming

- Find small gadgets in binary code.
- Compose gadgets together to accomplish work.

We can do this by placing the address of the next gadget to be called after the arguments to the current gadget. At the end of execution of the gadget, SP will point to the next gadgets address and machine will consider it as the caller and transfer control there by pushing the SP contents to IP. Gadgets have a small number of instructions one after another and end with a return, which can help identify them in a library's binary.

1.1 General Techniques for Arbitrary Computation

There are enough gadgets in libc to make a machine perform any arbitrary computation.

Consider the two gadgets that can load from an arbitrary address to a register and store to an address from a register. Load into $\%eax$, addr.

```
pop %eax
ret
```

Here the SP points to bytes containing the address of the argument. This address is popped to $\%eax$ register.

Store addr, $\%eax$

```
pop %edx
mov (%edx), %eax
ret
```

Here SP contains the address where contents of $\%eax$ must be stored. This gadget pops SP's contents to $\%edx$. To the address contained in $\%edx$ the contents of $\%eax$ are moved.

It is possible to string together gadgets in many libraries (to perform increment, decrement, loops, test equivalence, etc) to create machine code powerful enough to perform any computation.

1.2 Example Attack

- Use gadgets to call mmap.
- Call mmap to create pages in address space that are writeable and executable.
- Jump to the address created using mmap.

This helps an attacker overcome the NX bit that prevents executing code on the stack. In Just in Time compilers, the byte code is converted and written to pages in memory as machine code. Then the control is transferred to the translated pages and execution is performed. This is an example of a scenario where both Write and Execute permissions are enabled for a file.

2 Address Space Layout Randomization

It is a defense against all kinds of buffer flow attacks, such as

- Code Injection - By Stack Section randomization
- Return to libc - Code section randomization
- Return oriented programming - Can not locate gadgets within libraries at random locations.

2.1 Stack Randomization Technique

The position of the stack for a process is maintained in the Stack Pointer. SP is only incremented or decremented, but there is no instruction that causes SP to jump to a different value. Since a program just needs SP to keep track of the activation records, this technique moves the entire stack to different location and updates the stack pointer. This is done when the binary code of the process is loaded to memory, when the program is started and stack is setup. The stack's position does not change when a process is executing.

During code injection an attacker overwrites the return address of the vulnerable function to specify the address of the malicious program on the stack, to which the control should jump. Due to randomization, the address where process's stack begins and hence the address of malicious program changes during every execution, thus preventing execution of the intended malicious code with high likelihood.

When only stack randomization is supported, return to libc attack becomes more difficult but not unlikely. The argument to the function can not be passed as the address of the buffer in stack has been shifted due to randomization. One way to subvert this is to provide as input the intended attack string to a program that is known to use heap, several times. This fills the heap section (assumed to be shared among processes) and a random address is likely to point to the heap. Filling the string with spaces at beginning can help the attacker quickly succeed and carry out the attack.

2.2 Effective Address Space Layout Randomization

In addition to stack section randomization, the following are done to effectively implement address space randomization.

- Randomize location of all libraries (they must be page aligned).
- Randomize location of heap.
- Randomize mmaped location.

Randomizing the text section and location of libraries is effective in preventing return to libc attack and return oriented programming. In the later case, the attacker can not scavenge and look for gadgets to string together to execute arbitrary code as its position in the address space is not fixed.

2.3 Limitations of ASLR in 32 bit machines

- Moving around libraries that are forced to be page aligned may lead to fragmentation of virtual memory address space.
- In a 32 bit machine there is limited room for randomization (about 2^{16}) different starting addresses are only possible for a given section.
- Locating this may lead to several crashes that can alert a system administrator. It has been shown that the vulnerability can be exploited in about two hours in a 32 bit machine.
- 64 bit machines have large address spaces that can support ASLR and reduce significantly the success of an attack compared to 32 bit machines.

2.4 Limitations of ASLR in Windows OS

Linux supports position independent code (*pic*), that lets a process run a library code irrespective of its position in the address space. This was done to prevent the clash of libraries that hard-coded themselves to specific locations. This has been extended to support ASLR.

In case of Windows, there are different options for a user to choose from based on config options.

- ASLR on.
- ASLR Opt-in : Libraries declare in header to say if they can be loaded at different locations.
- ASLR Opt-out: Libraries declare in header to say if they opt out from ASLR.

Libraries that prevent ASLR and force the OS to load them at specific locations are good sources of gadgets that can be used to perform return oriented programming attacks. The lack of randomization enforcement has led to attacks on Adobe Acrobat Reader by exploitation of gadgets within libraries that must be loaded at a specific location in the address space.