

# **On the Effectiveness of Address Space Randomization**

CS 6V81 – Language Based Security

Jonathan Greene

4/7/2008

# Traditional Buffer Overflow Attack

```
#include<stdio.h>

int main()
{
    char buff[15];

    printf("Please type your name:\n");
    gets (buff);
    printf("Hello, %s", buff);

    return 0;
}
```

- Pseudo code for attack:  
Input EXEC("bin/sh") ptr2buff
- When the program tries to continue, it will execute the injected code, giving the attacker control of the system.

# Write or execute only memory pages

- Each page is marked with a bit that indicates whether the page may be written to or executed, but not both.
- Prevents maliciously injected code from executing since a buffer will be marked write only

# Return to libc buffer overflow attack

- Use C standard library functions like `system()` to gain control of a system
- Since every C program is linked to the C standard library, no code injection is necessary
- Steps for attack:
  1. Fill buffer with any data
  2. Over write return address with address of `system`, as well as the command we want system to execute
- Relies on knowledge of the address of standard library functions

# Obfuscation

## Normal Program:

```
#include<stdio.h>

int main()
{
    printf("On the first day of Christmas,\nmy
true love sent to me\nA partridge in a pear
tree.\n\n");

    printf("On the second day of Christmas,\nmy
true love sent to me \nTwo turtle doves,
\nAnd a partridge in a pear tree.\n\n");

    printf("On the third day of Christmas,\nmy
true love sent to me \nThree French hens,
\nTwo turtle doves, \nAnd a partridge in a
pear tree.\n\n");
}
```

## Obfuscated Program:

```
#include <stdio.h>
main(t,_a)char *a;{return!0<t?t<3?main(-79,-13,a+main(-
87,1-_
main(-86,0,a+1)+a)):1,t<_?main(t+1,_a):3,main(-94,-
27+t,a)&&t==2?_<13?
main(2,_+1,"%s %d %d\n"):9:16:t<0?t<-72?main(_t,
"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,
/#{l,+,/n{n+,/+n+,/#\
;#q#n+,/+k#,*+,/r : 'd* '3,}{w+K w'K:'+}e#';dq#'\ \
q#+'d'K#!/+k#;q#r}eKK#}w'r}eKK{nl}'/##;#q#n')}{#}w')}{nl}'/
+#n';d}rw' i;# \
){nl}!/n{n#'; r{#w'r nc{nl}'/##{l,+K {rw' iK;[{nl}'/w#q#n'wk
nw' \
iwk{KK{nl}!/w{%!##w#' i; :{nl}'/*{q#l'd;r'}{nlwb!/*de}'c \
;;{nl}'-{}rw}'/+,}##'*}#nc,'#nw}'/ +kd'+e}+;#rdq#w! nr/' )
}+}{rl#}'n' '# \
}'+}##(!/"/)
:t<-50?_==*a?putchar(31[a]):main(-
65,_a+1):main((*a=='/')+t,_a+1)
:0<t?main(2,2,"%s"): *a=='/' || main(0,main(-61,*a,
"!ek;dc i@bK'(q)-[w]*%n+r3#l,{:}\nuwloca-O;m
.vpbks,fxntdCeghiry"),a+1);}
```

# PaX: Address Space Layout Randomizer (ASLR)

- Since return to libc attack relies on knowledge of function address, initialize a program's address space so that an attacker must guess the function's location

# Anatomy of a process's address space

- Divided into 3 sections:
  1. Executable – instructions and data
  2. Mapped – heap, **linked libraries**, etc.
  3. Stack
- Each section is given a different, random, offset. The executable and mapped sections have 16 bit offsets, and the stack a 24 offset
- For a return to libc attack, the only area that matters is the mapped area since that is where the standard C functions reside

# Mapped addresses

- Limited to 16 bits (65,536) values because modifying top 4 or bottom 12 bits would cause page addressing problems
- This small number of values makes a brute force attack feasible
- The address of a library function in a PaX randomized process can be computed as:

$$\text{address} = 0x40000000 + \text{offset} + \text{delta\_mmap}$$

where

- 0x40000000 is the standard base address for mapped memory
- Offset is a value within the library for the desired function
- Delta\_mmap is the random offset generated by PaX

# Target System

- No known buffer overflow vulnerabilities in current version of Apache web server
- Recreated vulnerability found in Oracle 9 PL/SQL (real vulnerability was a 1000 char buffer, recreated one is 64)
- Brute force attack was successful with an average time of 216 sec (29 min, 810 max)

# Return to libc attack, stage 1

- Uses that fact that when a process crashes, a child process is forked with identical address offsets
- Attempt to guess the `delta_mmap` by overflowing the buffer, writing the guessed address of `usleep()` over the return address, and pass an argument of 16 seconds
- If the connection hangs for 16 seconds, `usleep()` has been found, and `delta_mmap` determined, and if it crashes immediately, increment `delta_mmap`, and guess a new address for `usleep`

top of stack (higher addresses)
⋮
<code>ap_getline()</code> arguments
saved EIP
saved EBP
64 byte buffer
⋮
bottom of stack (lower addresses)

Figure 1: Apache child process stack before probe

top of stack (higher addresses)
⋮
0x01010101
0xDEADBEEF
guessed address of <code>usleep()</code>
0xDEADBEEF
64 byte buffer, now filled with A's
⋮
bottom of stack (lower addresses)

Figure 2: Stack after one probe

# Return to libc attack, stage 2

- Once `delta_mmap` is determined, the location of `system()` is known
- Overflow the buffer one more time, this time, filling the buffer with shell commands. Overwrite the return address with the address of `sleep()` and followed by a series of `ret` instructions

top of stack (higher addresses)
⋮
<code>ap_getline()</code> arguments
saved EIP
saved EBP
64 byte buffer
⋮
bottom of stack (lower addresses)

Figure 3: Apache child process stack before overflow

top of stack (higher addresses)
⋮
pointer into 64 byte buffer
0xDEADBEEF
address of <code>system()</code>
address of <code>ret</code> instruction
⋮
address of <code>ret</code> instruction
0xDEADBEEF
64 byte buffer (contains shell commands)
⋮
bottom of stack (lower addresses)

Figure 4: Stack after buffer overflow

# Other attacks

- Certain buffer overflow attacks can be converted into formatted string attacks, e.g. `printf(buffer)` instead of `printf(“%s”, buffer)`.
- Using a formatted string attack, the value of `delta_mmap` can be determined more quickly

# Improvement 1: 64 bit architecture

- On a machine with 64 bit addressing, address space randomization **is** an effective deterrent to attackers because up to 40 bits can be randomized (>100 years on average vs. 2<sup>16</sup> sec)
- Not a feasible solution because a 32 bit program running in compatibility mode retains threat

# Improvement 2:

## Increase randomization frequency

- One advantage attacker had is the child process is forked with the exact same random offsets as the part, making a sequential brute force attack effective.
- Expected number of probes to determine delta\_mmap:

$$\sum_{t=1}^{2^n} t \cdot \frac{1}{2^n} = \frac{1}{2^n} \cdot \sum_{t=1}^{2^n} t = (2^n + 1)/2 \approx 2^{n-1}.$$

- If re-randomizing after each probe, the expected number of probes is only increased by a factor of 2

# Improvement 3:

## Randomization Granularity

- By randomizing at compile time, an additional 10-12 bits of randomness can be generated (making a brute force attack infeasible), however, since libraries are shared, if location is leaked from another part of the system, the added benefit is eliminated.
- By increasing runtime randomization to 20 bits, memory paging becomes more difficult, and increasing randomization by 4 bits does make a brute force attack infeasible
- Reordering functions is also not effective because the attack can be redesigned so that the `system()` function is the only function that needs to be found.

# Defending against the attack

- There is no effective way to “watch” for an attacker and stop them, since this requires a quick response, and the daemon being taken offline
- The most effective defense against this attack is to defend against stack overflow, which is a technique independent of Address Space Randomization

# References

- Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. [On the Effectiveness of Address-Space Randomization](#). In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, October 2004.
- The PaX Team. [PaX Documentation: Address Space Layout Randomization](#). <http://pax.grsecurity.net/docs/aslr.txt>, May 2003.