

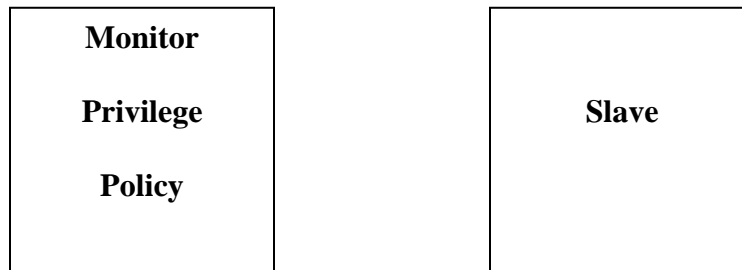
# CSE509 Lecture Notes for 03/30/06

Pramod Adiddam

## **Privilege separation:**

### **Policies**

Privilege separation is to be secure. In privilege separation, there are two components -  
Privilege program = Privilege + Policy  
where, privilege and policy are usually bound together.



Hence, it is required for the policy to go into the monitor because the privilege would be in the monitor.

Also, an ideal privilege separation wouldn't introduce new bugs. That is, new separated program should have a bug if and only if original program has a bug. Original program may have many bugs. These bugs will be carried intact to either the monitor or slave. Such a bug can land in slave or monitor. If it lands in the slave (and the privilege and policy are in the monitor) then it is OK to be in such a situation because now that bug can't be exploited to gain privileges. If the bug lands in the monitor, then it is not clear whether it is easy or difficult to exploit it as it might be the case that now there is less code to find the bug or the bug can now be difficult to reach.

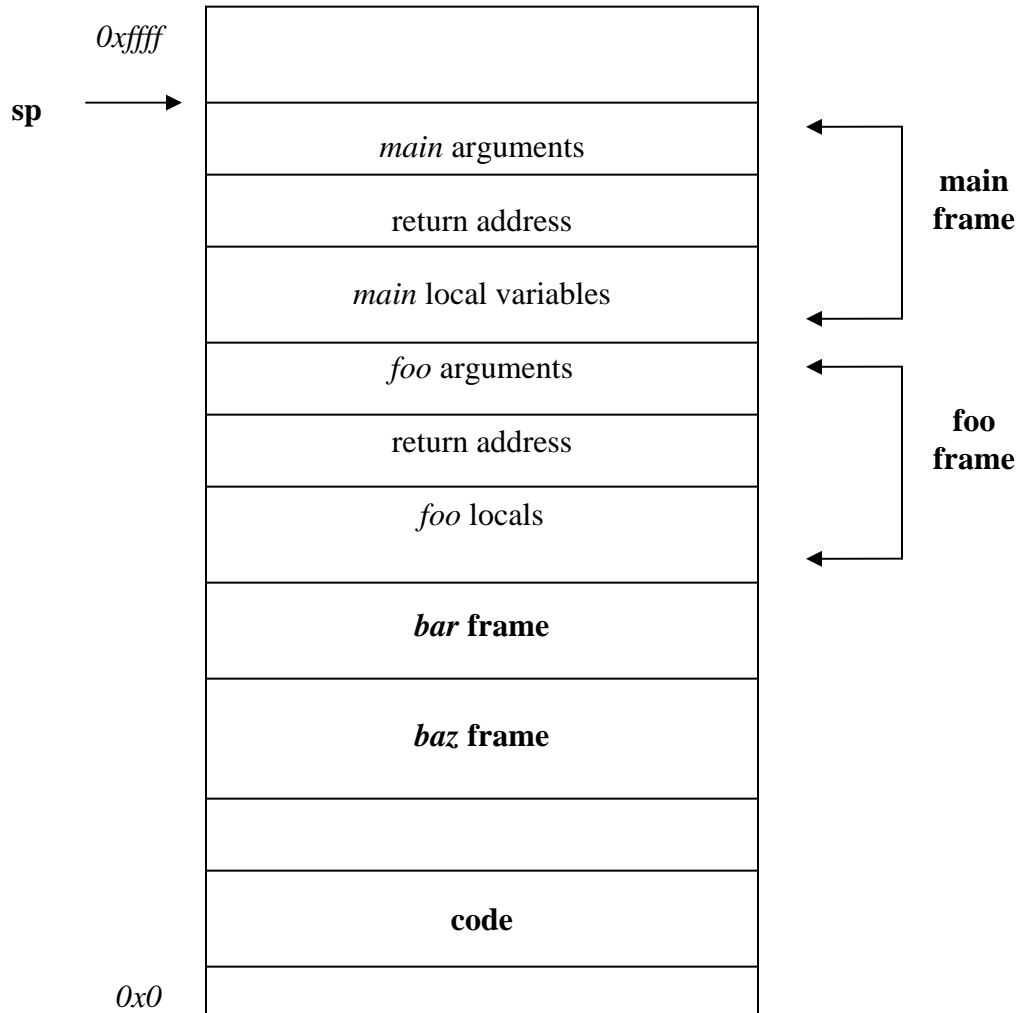
## **Buffer overflows:**

Let us see where the attacker can be stopped by going through a list of what an attacker would do:

1. Find an overflow in source – (These kind of attacks are countered by tools like BOON)
2. Send overflowing message – (Tools like CCured)
3. Overwriting the return address – (by Address Space Randomization)
4. Program returns to Return Address
5. Execute shell code
6. Makes system call

## Randomization

For a buffer overflow, attacker needs to get the address of the buffer. If he can get the exact copy of the program that will run, then he can do that by running a debugger. If we make it hard to find address of the buffer then we are fine. One way is to randomize the top of the stack. Return address is absolute but the location of the return address in the stack can be computed by the size of the arguments, local variables etc.



*The Stack*

*main* calls *foo* which decrements the  $sp$  by some number. Each new decremented  $sp$  is relative to the previous  $sp$  and hence changing the stack top every time a process starts, is going to change the  $sp$  that is stored in return address. Here attacker cannot be prevented from accessing the buffer and hence the return address (RA) and can only be prevented from entering the absolute address in RA. This solution only prevented code injection in buffer accessed from RA written by the buffer overflow.

## More randomization for other kinds of attacks

- randomize stack top
- randomize *libc* location
- randomize code location (recompile the code to not use absolute addressing or one shot randomization by rewriting the binary)
- randomize data location
- interframe padding (code needs to be invasively modified to insert padding in between the local variables etc. and return address)
- rearrange function (can be done with an added level of indirection in the code for *jmp* to the functions)
- rearrange basic blocks
- rearrange local variables (the last 3 including this make it real difficult for return to code attacks)
- rearrange data
- randomize *mallocs* (for preventing exploitation of the serial order of allocation by *mallocs*)

Not all of the above make it difficult for any attack. Each of them mostly is attack particular. As discussed in the review paper, on a 32-bit system, address space randomization makes the attacker guess 22 bits to attack the system. It is shown that it is easy for such a guess to find the code and get in. Solutions include using 64-bit etc.

## Pointguard

Most of the solutions discussed can be made worse by a format string attack previous to the buffer overflow attacks.

```
void foo (char *c)
{
    char buf [1024];

    strcpy (buf, c);
    return;
}
```

We can see above that buffers are usually *chars* or *ints*. RA is a pointer type and buffer is not. We don't use pointer types for buffers since we usually do not read in pointers into a buffer. So if the buffer is tagged (the RA is tagged as *p* and the buffer with its type in the memory. So these tags will be overwritten with incoming data type every time there is a memory write to a tagged location. So if we write *int* buffer overflow to RA, then while loading RA we see that it is not a pointer type and raise an exception, possibly. So only a pointer RA is used to *jmp*.

Question is where to store the tag?

Another idea is pointers are stored in the memory encrypted and *int/ chars* etc. are not. At startup pick a key *k* and encrypt all the pointers. So overwriting will

give garbage upon decryption. To make encryption/ decryption faster, XOR is the encryption method used.

If the key is in register it is the safest bet. But if for some performance/ processor limitation reason we can't store the key in the registers, it can be stored in the memory and can be read. By a format string error, we can print out an encrypted RA and XOR it to the RA (which we know already) to get the key back.

Another method is to use a bigger landing pad (discussed in previous classes) to make the last bits of the buffer address redundant and hence make it easier to guess the address.

## **Stackguard**

Problems with recompiling to separate buffer and RA include problems across old and new libraries. Hence here the idea is; at startup of the process pick random *canary* (a 32-bit number) and write the *canary* below RA. So *canary* cannot be guessed and if overwritten by a buffer overflow (that overwrites the RA too after overwriting the *canary*) we can catch the buffer overflow by comparing it to the *canary* and check before the RA is loaded.