

Overflow situations, protections, and exploit techniques

Centinel Hackfest 2003

Outline

Overflow Locations

- Stack
- Heap

Overflow Situations

- String Functions
- Integer Overflows
- Signed Issues
- Calculations
- Format Strings

Outline

Overflow Prevention

- StackGuard
- Non-Exec Stack
- Non-Exec Heap
- W^X / ProPolice / OpenBSD
- PaX / grsecurity / linux

Outline

Hacking Like A Rockstar

- Core Impact
- InlineEgg
- Canvas
- Mosdef
- Metasploit Framework
- Mad Secret Stuff

Overflow Locations

Da Stack

Used for storing local function data

```
int fun(int i) {  
    char array[4];  
}
```

`array` would be located on the stack

Da Stack

Used for function calls

Address of where to return to is pushed onto the stack in the call instruction

This address is popped from the stack and jumped to in the ret instruction

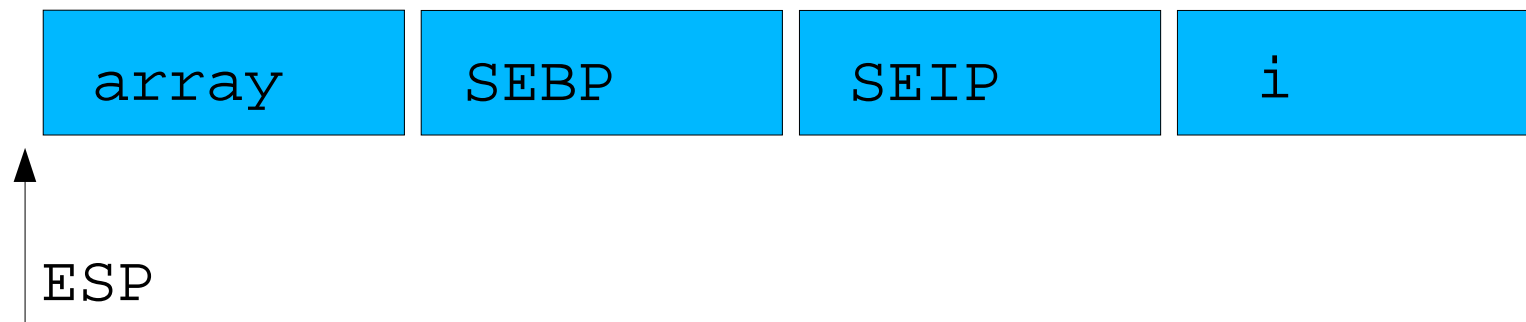
Overwriting this stored address is good

Da Stack

Generally in the function's prolog, EBP is also pushed onto the stack and restored in the epilog

Typical stack layout during function

Previous fun() function's stack

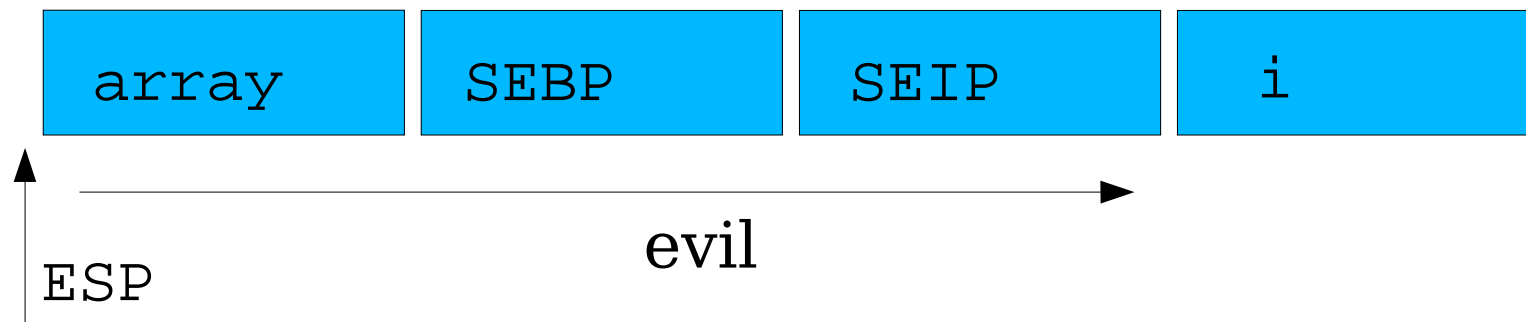


Da Stack

If we can cause an overflow somewhere and write into the saved `EBP` and more importantly `EIP`, we can do evil.

This is the most common way to gain control of program execution

Previous `fun()` function's stack



Da Heap

The heap is the area of memory that is dynamically allocated through `malloc` family of functions

“new” in C++ allocates here

Since memory in this area is dynamically located, there is often a structure to the heap, which can lead to the ability to manipulate memory

Da Heap

```
int fun() {  
    char *ptr = (char *)malloc(100);  
}
```

Depending what else is on the heap,
overflowing may allow you to
overwrite useful things (function
pointers, etc)

Da Heap

Doug Lea's Malloc

dlmalloc is a heap structure used by linux and others, organizing the allocated memory in a doubly linked list

If you can cause an overflow, you can write a fake entry and use the pointers of this linked list to write into memory

Da Heap

What good is writing 4 bytes?

- Function Pointers
- Return Address
- `__free_hook` family
- Global Offset Table

This is a complex situation with lots of different techniques, check out phrack for more info.

Da Heap

A brief note about win32

Heap overflows are more common in windows, and often contain more influential data like function pointers. Also have SEH and other structures in the heap

Mostly C++ code is part of the reason for this

Overflow Situations

String Function Overflows

This is you most common overflow

- `strcpy`
- `strcat`
- misuse of `strn` functions

String functions stop at a null byte, so you if this is in your overflow data, the overflow will stop

Signed Issues

Checking a number like it is unsigned
when in fact it is not

```
if(len > sizeof(buf)) {  
    error();  
}  
else {  
    strncpy(buf, userbuf, len);  
}
```

Integer Overflow

Two numbers independently are smaller than the largest unsigned int, but together in a calculation cause an overflow and become a small number

```
malloc(sizeof(struct) * number);
```

Allocate too little space for what we will end up writing. Overflow.

Calculation Problems

Be careful doing math!

- Pointer arithmetic
- `strn` family of calls
- Function calls like `read`

It is common to check if a value is too large, but not if it is too small

Calculation Problems

Real World: PoPToP Vulnerability

Bugtraq excerpt:

```
....  
if (length > PPTP_MAX_CTRL_PCKT_SIZE) {  
    // abort  
}  
....  
bytes_this = read(clientFd, packet + bytes_ttl, length - bytes_ttl);
```

If given length was 0 or 1, the "length - bytes_ttl" result is -1 or -2, which means that it reads unlimited amount of data from client into "packet", which is a buffer located in stack.

The exploitability only depends on if libc allows the size parameter to be larger than SSIZE_MAX bytes. GLIBC does, Solaris and *BSD don't.

Formatting Strings

- Channeling Problem
- User defined formatting string

```
printf (userbuf ) ;
```

```
syslog (LOG_ERR , userbuf ) ;
```

Formatting Strings

- Multiple Interpolation

```
sprintf(buf, "%s", userbuf);  
printf(buf);
```

Formatting Strings

- DoS Program

```
printf(“%s%s%s%s%s ...”);
```

- Read Stack Contents

```
printf(“%08x%08x ...”);
```

- Write To Memory

```
printf(“AAAA%08x%n”);
```

Overflow Prevention

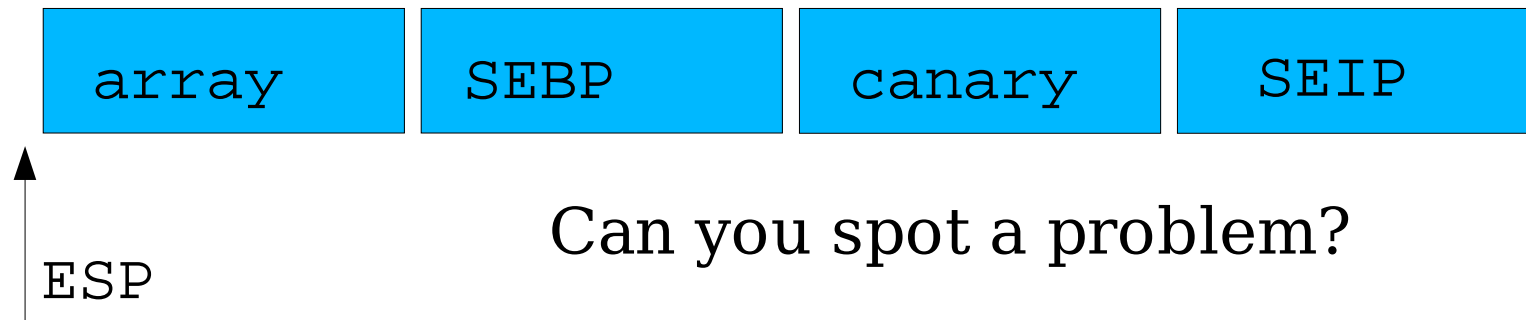
(and of course circumvention of them)

StackGuard

I stop overflows! (Well, kinda)

- Uses stack canaries to detect if the SEIP has been overwritten.
- Several canary generation techniques, including reading from `/dev/urandom`

Previous fun() function's stack

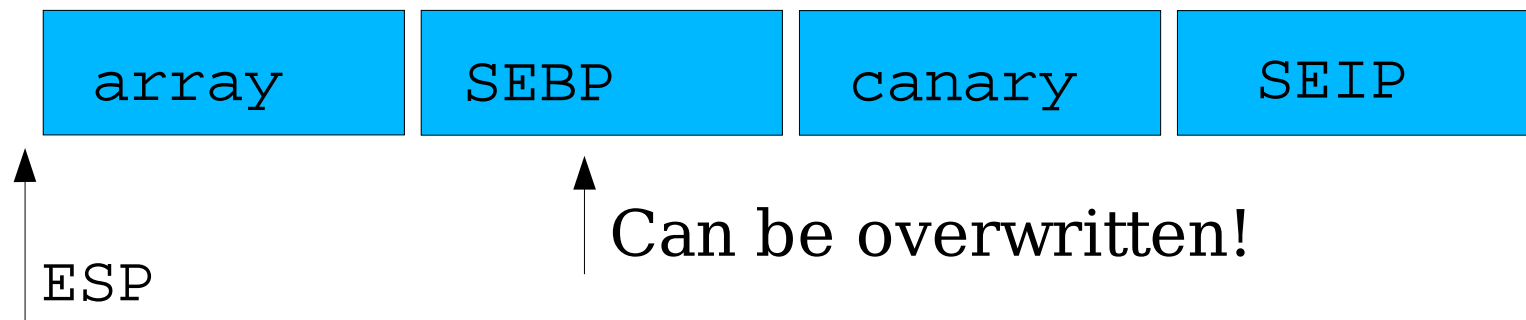


StackGuard

Several problems

- Canary value does not protect SEBP
- Does not protect other data on the stack before SEIP (pointers, etc)

Previous fun() function's stack



Non Executable Stack

- Stack is flagged non executable
- We cannot run our code here.

All is not lost!

- We can still own `SEIP`, and gain control
- Put shellcode elsewhere (heap)
- This isn't very much protection

Non Executable Heap

- Heap is flagged non executable
- We cannot run our code here

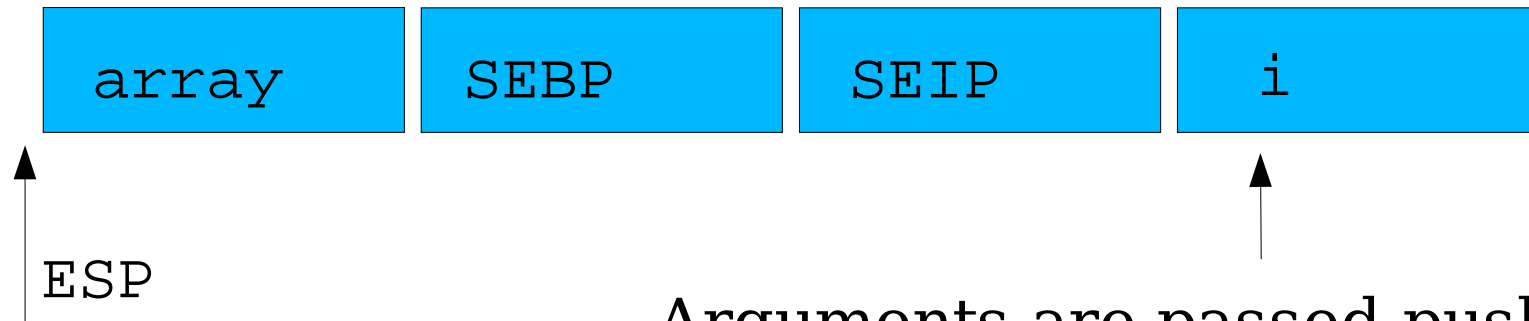
Damn...

- No place to stick our code
- Have to work with code already present in the program

Non Executable Heap

Return To Libc

Previous fun() function's stack



Arguments are passed pushed onto the stack before the call

Non Executable Heap

If we overwrite `SEIP` with the address to a function, it will be like a `jmp`

The function will expect the `ret` address and arguments on the stack

Previous `fun()` function's stack



Non Executable Heap

- Overwrite `SEIP` with address of `system`
- Overwrite `arg1` with pointer to string
- `/bin/sh` and others are already in `libc`!
- If you want a different string, put it in the process (environment, stack, heap, etc)

Previous `fun()` function's stack



↑
ESP after return

W^X / ProPolice / OpenBSD

ProPolice

- Gcc extension, only protects code compiled with it
- Puts a canary before SEBP
- Moves pointers before buffers
- Doesn't protect the heap
- Protects against Return To Libc

W^X / ProPolice / OpenBSD

W^X

- Page is either writable or executable
- Can't execute in the stack or heap
- Return To Libc still works (without pp)

W^X / ProPolice / OpenBSD

OpenBSD (newer versions)

- It's got W^X
- It's got ProPolice
- It also randomizes library mappings (libc) to make return to libc impractical
- Solid solution, I doubt to see many exploits
- Kernel bugs still allow great things
- OpenBSD also has systrace!
 - Syscall rulesets for processes

PaX / grsecurity / linux

PaX

- Basically the same as W^X
- Does a lot of randomizations of address basings (stack, heap, etc)
- Return To Libc possible but improbable

PaX / grsecurity / linux

grsecurity

- It does too much awesome stuff to list here
- ACL's, systrace like control of processes
- Insane logging
- Randomization of everything (PIDs, etc)
- Includes PaX
- Good Solution, doubt to see any exploits that circumvent it

PaX / grsecurity / linux

User Mode Linux

- Exploit mitigation not prevention
- Run a whole linux system as a user
- Compartmentalize!

Hacking Like A Rockstar

Ok....

Why is this framework crap important

- All modules share payloads (shellcodes)
- Access to encoders
- Multistage shellcodes made easier
- Library functions that make exploit development faster and more robust
- Ability to do stuff like proxying, ssl, etc

Ok....

Why is this dynamic shellcode stuff important

- Make a payload on the fly before exploitation
- Allows for easy generation of things with hardcoded but user defined strings
- When doing syscall proxying, this can make things a lot easier
- Even if you don't use it dynamically, it still makes shellcode development nicer

In case you don't know

Injection vector – Getting the shellcode into the process and getting the process to the point of jumping control to the shellcode

Shellcode/Payload – The actual assembly instructions that do the cool stuff, lots of possibilities (adduser, connect back, `rm -rf ;`))

In case you don't know

Syscall – Call kernel code to do a job for you that you would otherwise not have permission to do. Pretty much any call that leaves process (read, write, exec)

Syscall Proxying – A syscall server reads in a stack image and function pointer/syscall number from the syscall client (via the network). Recreates the stack image and then jumps to the pointer or calls the system call. **No Code** uploaded or executed. Stack after call is returned to client

Core Impact

Exploit Framework

- Exploit modules written in python
- Syscall proxy server/client
- Supports “pivoting”
- Some of the best guys in computer security work on it (gera, etc)
- Ekk, ~27k USD

InlineEgg

Dynamic Shellcode

- Written in python
- Allows you to dynamically generate shellcode in a decent programmatic way
- Syscalls, loops, variables, all made easy
- Restrictive license

Canvas

Exploit Framework

- Dave Aitel is a ninja and a half
- Syscall proxy server/client
- Modules and framework written in python
- Module dev isn't as friendly as it could be
- Solid modules
- Only 1k USD

Mosdef

C -> Assembly compiler

- Python compiler
- Write c-ish code, get assembly
- Dynamic shellcode
- Released under GPL
- No documentation

Metasploit Framework

Exploit Framework

- mmm.... perl
- HDM Rocks (and he writes great modules)
- Writing modules is unpainful
- Free, Open Source, Community
- Great library, Encoders, random nops, etc
- Socket library (future ssl, proxy support)
- Access to payload collection (future support for dynamic payloads from any executables)

Hacking Like A Rockstar

Mad Secret Stuff

(too secret for slides)