Exploits

Cody Cutler and Anton Burtsev
Malware Analysis Seminar, 2012
Roadmap

- Bypassing stack protection
- Bypassing execute protection
- Bypassing ASLR
- Heap spraying
- JIT spraying
- Heap exploits
Bypassing stack protection

Stack Smashing Protector
Paul Rascagneres
Stack protection

Canaries placed before EIP

Canaries are verified in function epilogue

Reorder local variables

Reorder arguments
Bypassing stack protection

Hard to overflow EIP if implemented correctly

Exploit information leaks to discover canary value

Overwrite canary early
Bypassing execute protection

Bypassing non-executable-stack during exploitation using return-to-libc context
Execute protection

The hip buzzword is DEP (Data Execution Prevention)

RET'ing to memory (stack) with execute permissions disabled faults

No more RET'ing to our shellcode :(
Bypassing execute protection

Program and libraries are loaded at constant addresses

Overwrite EIP with system() and continue overflowing to provide arguments; "/bin/sh" etc.
Bypassing ASLR

Bypassing PaX ASLR protection
Phrack, Tyler Durden

On the Effectiveness of Address-Space Randomization
Hovav Shacham (Stanford), et. al.
ASLR

Address space layout randomization

Program segments, dynamic linker, and libraries are loaded at random* base offsets

Code must be position independent

What do we overwrite EIP with?
Bypassing ASLR

Loaded at random base address, but code is still one segment

Useful code nearby?

Overwrite LSBs of old EIP after overflowing

Repeat the overflow and exploit information leakage
Bypassing ASLR

Example from Tyler Durden's Phrack paper
main(...

int do_auth()
{
    printf("Password: ");
    fflush(stdout);
    len = read(0, pass, sizeof(pass) - 1);
    if (len <= 0)
        FATAL("read");
    pass[len] = 0;
    if (!verify(pass)) {
        printf("Access granted .\n");
        return (0);
    }
    printf("You loose !");
    fflush(stdout);
    return (-1);
}
__libc_start_main(...
main(...
do_auth(...
int verify(char *pass)
{
    char filtered_pass[32];
    int i;

    bzero(filtered_pass, sizeof(filtered_pass));

    /* this protocol is a pain in the ass */
    for (i = 0; pass[i] && pass[i] != NL && pass[i] != CR; i++)
        filtered_pass[i] = pass[i];

    if (!strcmp(filtered_pass, OKAY_PASS))
        return (0);

    return (-1);
}
Disassembly

(...)
8048595:  68 bc 86 04 08       push  $0x80486bc
804859a:  e8 5d fe ff ff       call  80483fc     <printf>
(...)
80485f4:  e8 27 ff ff ff       call  8048520     <verify>
80485f9:  83 c4 10           add   $0x10,%esp
Bypassing ASLR

Overwrite last byte of EIP to change from 0xf9 to 0x9a

Enter tricky string to coerce the printf to leak tasty infos

Print arbitrary, previous pushes on stack
"%1$08x"
"%2$08x"
"%i$08x"
Bypassing ASLR

Turns out `__libc_start_main(..` is called with argument `main=<main() address>`

Also turns out `__libc_start_main()` is a constant offset away from useful things like `strcpy()`, `system()`, and `setreuid()`
Bypassing ASLR

Putting it all together: Overwrite LSB of EIP to point to near printf providing a format string to print the "main" argument to \_\_libc\_start\_main()

Add precalculated offset to this address to get absolute address of useful functions

Overwrite EIP with this new address when we arrive at exploit again

See paper for more depth
Bypassing ASLR

In Hovav Shacham's paper:

32 bit arch don't offer enough bits of entropy

Parent and forked children are not randomzied (PaX)

They still exploit Apache overflow in under 300 seconds
Heap spraying

Even with ASLR, some allocators are predictable

Make host put shellcode all over heap (variables in interpreters)

Overflow EIP to heap

DEP complicates things
Heap spraying

var evil = "\x90\x90\x90\x90\x90......
<evil shellcode>"

bigarray = new Array();
for (c = 0; c < 0x1000; c++)
    bigarray[c] = evil;
JIT spraying

Interpreter Exploitation: Pointer Inference and JIT Spraying
Dion Blazakis

JIT Spraying and Mitigation
Piotr Bania
JIT spraying

Just in time compilers are great!

By definition, they create executable code (no DEP)

Instead of jumping to *contents* of variable, jump to *code* that creates the variable

Spray it to mitigate ASLR
JIT spraying

Initialize a variable using math operators

XOR is nice (ActionScript):
  one-to-one from script to bytecode
  bytecode side effects are semantic no-ops
  provides four bytes for immediate

var evil = 0x3c001d0d ^ 0x3c0dbeef ^ ....
JIT spraying

This give us:

mov %eax, 0x3c00ld0d
xor %eax, 0x3c0dbeef
xor %eax, ......

etc.
JIT spraying

Careful selection of immediates and disassembling from [code address + 1] yields:

Dion Blazakis' example:

```
FNOP
PUSH ESP
CMP AL,35 ; MSB of imm 0x3c and XOR opcode = semantic no-op
POP EAX
NOP
NOP
CMP AL,35
PUSH -0C
POP ECX
CMP AL,35
ADD EAX,ECX
NOP
CMP AL,35
FSTENV DS:[EAX]
```
JIT spraying

Once again, overflow EIP with code address

Finding the address is tricky, see paper for details
Heap exploits

Ran out of time :(

Overflow into per-block allocator data and be tricky