Buffer Overflow Attacks

History

Amongst the many different types of vulnerabilities which exist, Buffer Overflows (Overrun) still persist

Date back to the 1980's

In practice, a large amount of overflow techniques seen are accounted for

There is a constant evolution of methods, attacks refined

Defences might have or introduce a weakness

Cycle continues (patch, see new exploit)

History

Most, if not all Buffer Overflows are the result of buggy C code

No length checking on the buffer

C doesn't inherently check the length of buffers

A potential solution?

Don't use C

Languages that do check:

Java, Python, ...

Not So Easily Avoided

What language is Python written in?

- **C**

What language would system libraries being called be written in?

- **C**

Which are running on an OS, what language is the kernel written in?

- **C**

✓C however is popular language

✓ Low level system control

- Code reuse
- Poor practice
- Mistakes introduced

What is an Exploit?

- An exploit is any input (i.e., a piece of software, an argument string, or sequence of commands) that takes advantage of a bug, glitch or vulnerability in order to cause an attack.
- An attack is an unintended or unanticipated behavior that occurs on computer software, hardware, or something electronic and that brings an advantage to the attacker.

What is an Exploit?

- An exploit is not necessarily a program.
- While it can be a program that communicates bad input to a vulnerable piece of software, it can also be just the bad input itself.
- Any bad input (or even valid input that the developer just failed to anticipate) can cause the vulnerable application to behave improperly.

Buffer Overflow Attack

- One of the most common OS bugs is a buffer overflow
  - The developer fails to include code that checks whether an input string fits into its buffer array.
  - An input to the running process exceeds the length of the buffer.
  - The input string overwrites a portion of the memory of the process.
  - Causes the application to behave improperly and unexpectedly.

Buffer Overflow Attack

- Since the stack grows downward, if you write past the end of the buffer, you can corrupt the content of the rest of the stack, if enough information is known about the program.
- Because of the nature of the address space, locally declared buffers are allocated on the stack and one could write over known register information and the return address.

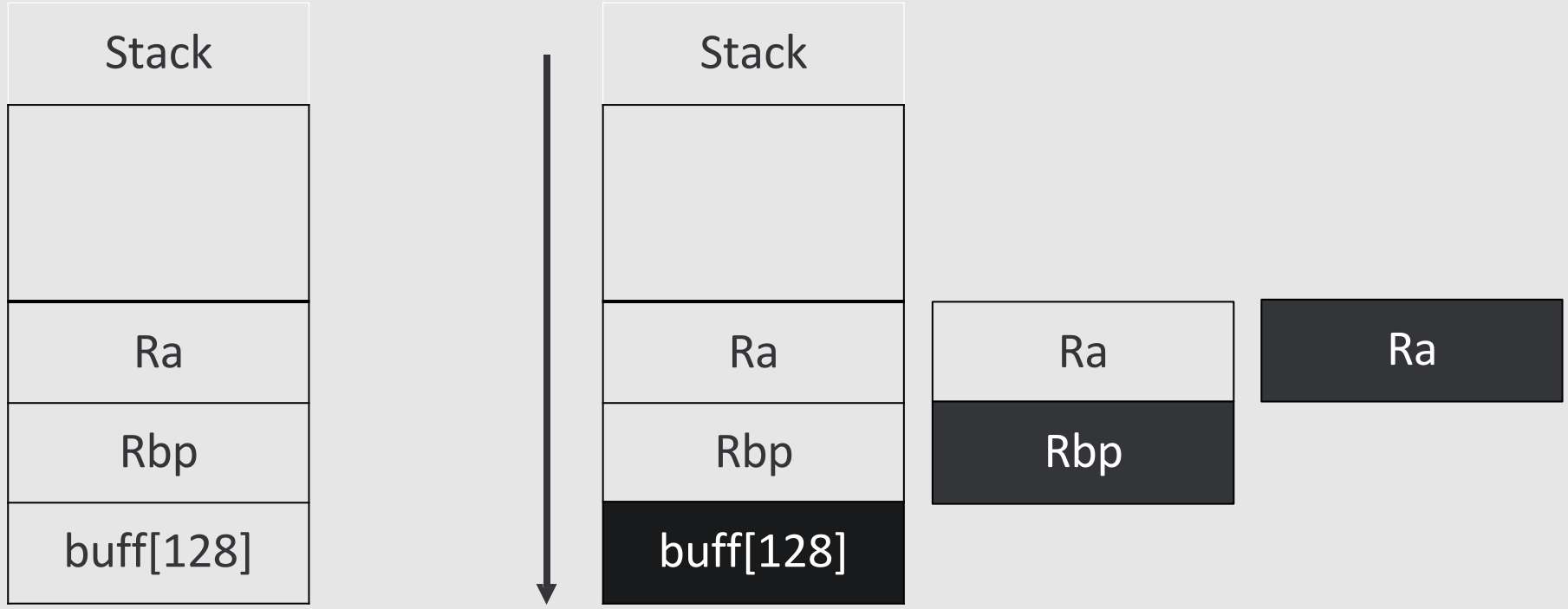
Buffer Overflow Attack

- Effect of a buffer overflow
  - The process can operate on malicious data or execute malicious code passed by the attacker.
  - If the process is executed as root, the malicious code will be executing with root privileges.

Control

popfunc()

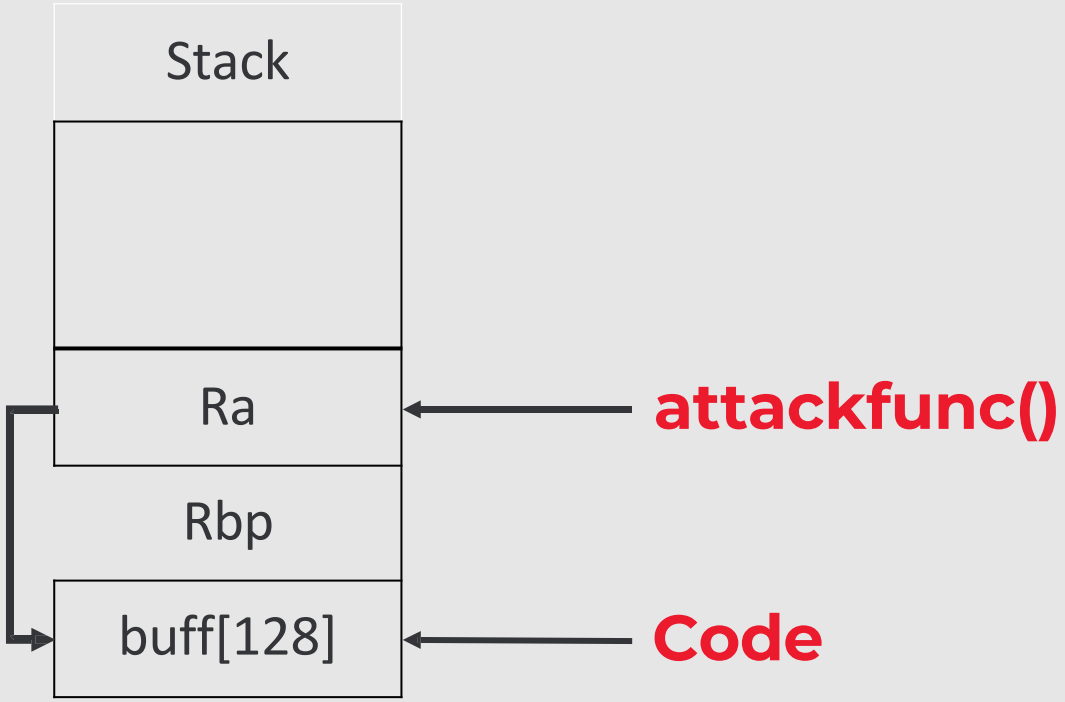
Ra: Return Address
Rbp: Frame pointer
Buff: Allocated buffer



Attacker chooses what goes in to function

Control

popfunc()



Overflow the buffer

Function returns

Tidy up Ra, whatever

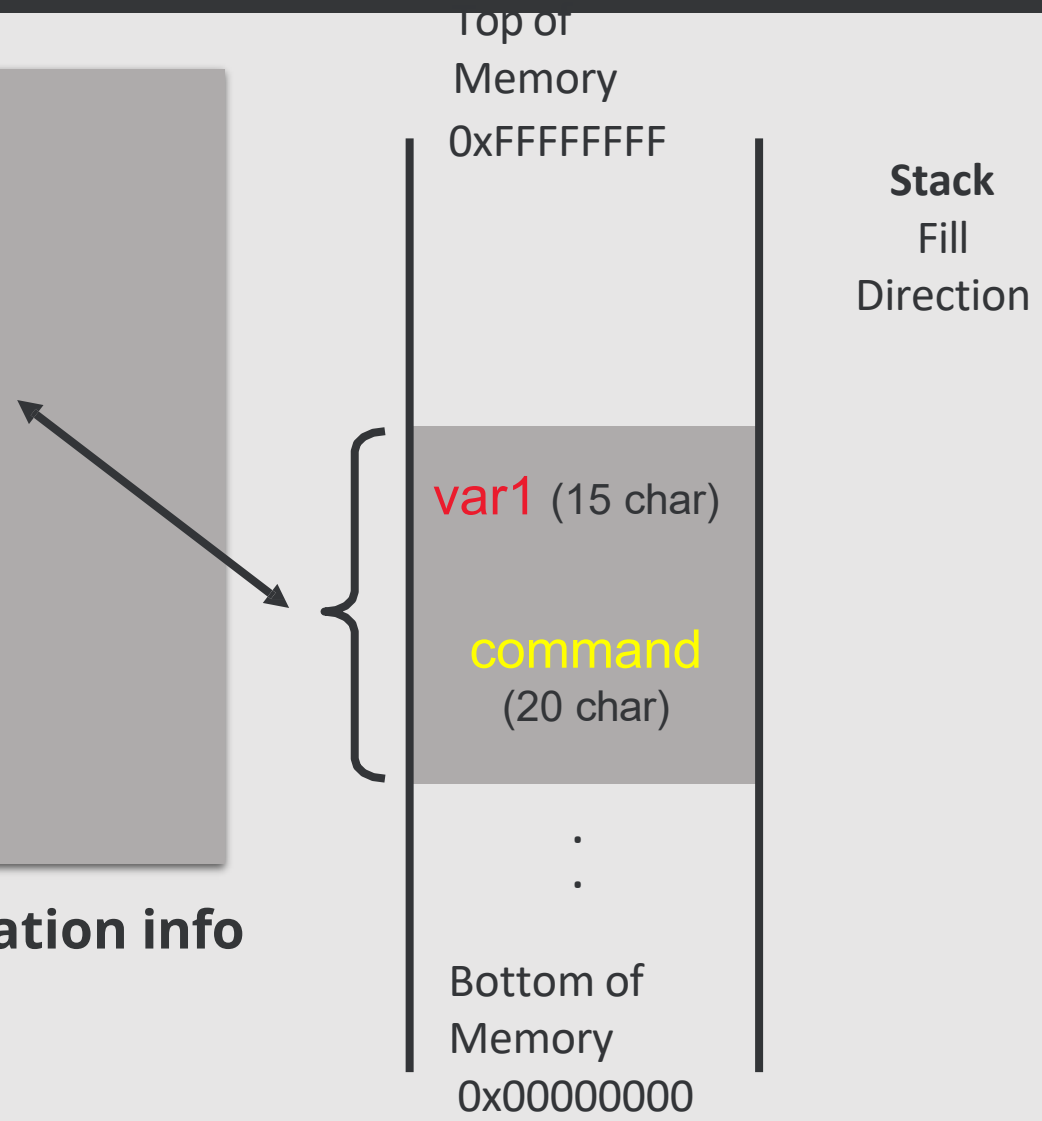
Open Source programs
Github

Assume attacker has access to the source code

Buffer Overflow

domain.c

```
main(int argc, char *argv[ ])
/* get user_input */
{
   char var1[15];
   char command[20];
   strcpy(command, “whois ");
   strcat(command, argv[1]);
   strcpy(var1, argv[1]);
   printf(var1);
   system(command);
}
```

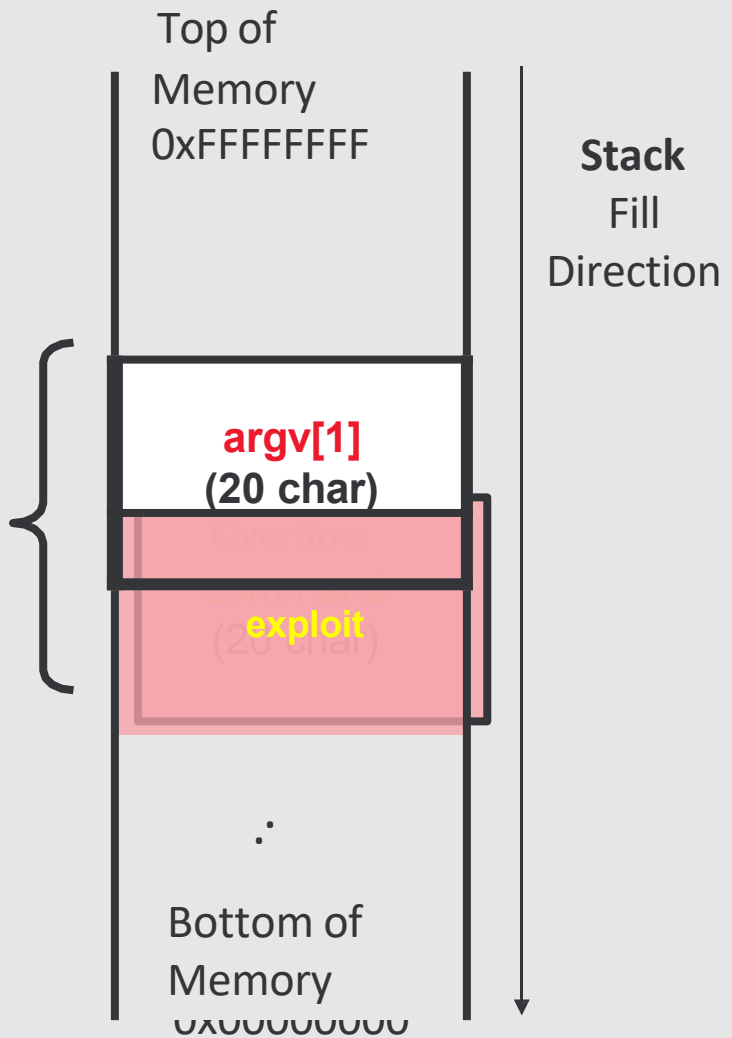


Retrieves domain registration info
e.g., domain swin.edu.au

strcpy() Vulnerability

domain.c

```
Main(int argc, char *argv[])
/*get user_input*/
{
   char var1[15];
   char command[20];
   strcpy(command, “whois ");
   strcat(command, argv[1]);
   strcpy(var1, argv[1]);
   printf(var1);
   system(command);
}
```



- **argv[1] is the user input**
- **strcpy(dest, src) does not check buffer**
- **strcat(d, s) concatenates strings**

strcpy() vs. strncpy()

- **Function strcpy() copies the string in the second argument into the first argument**
  - e.g., strcpy(dest, src)
  - If source string > destination string, the overflow characters may occupy the memory space used by other variables
  - The null character is appended at the end automatically
- **Function strncpy() copies the string by specifying the number n of characters to copy**
  - e.g., strncpy(dest, src, n); dest[n] = '\0'
  - If source string is longer than the destination string, the overflow characters are discarded automatically
  - You have to place the null character manually

Stack-based BoF detection using a random canary

Normal (safe) stack configuration:

Buffer overflow attack attempt:

- The canary is placed in the stack prior to the return address, so that any attempt to over-write the return address also over-writes the canary.
- The system regularly checks the integrity of this canary value. If it has been changed, it knows that the buffer has been overflowed and it should prevent malicious code execution.

Testing

Code must be carefully inspected and tested to discover all possible buffer overflows.

Code Inspection

Expensive and time consuming.

Fuzz Testing

Input random strings into input variables and data files.

Fire random events while a process is running

Easy to automate

Log input, output, behaviour to discover potential vulnerabilities.

Used by crackers to discover vulnerabilities too!

Preventing Stack-based BoF Attacks

- PointGuard (Microsoft). It is a compiler extension, that address code which XOR-encodes any pointers, including the return address before and after they are used. Therefore, attacker cannot reliably overwrite the return address.
- Data Execution Prevention (DEP) tags memory as Read-Execute (code) or Write-NoExecute (data), enforces no-execution permission on the stack segment of memory. Attackers responded by using ROP.
- Address space layout randomization (ASLR) rearranges the data of a process's address space at random. Defeats ROP.

Data Execution Prevention

In an attempt to stop buffer overflow exploits, CPU manufacturers have created DEP.

http://support.microsoft.com/kb/875352

DEP is a feature which allows particular areas of memory to be tagged as NX – non-executable (AMD) or XD – execute disabled (Intel).

Address Space Layout Randomization

- Aims to make ROP impossible
- ASLR is a scheme of changing the layout of the heap, stack, and library functions after each boot to make it harder for a hacker to locate areas of useful memory.
- It is available since Vista/7 and in Linux kernels since 2.6.2.

Return-oriented programming

- ROP
- Locate useful portions of code (system calls) at the end of existing OS .dlls (already loaded into memory).
- Write exploit which jumps to each useful command in the correct sequence to perform a malicious act.
- Avoids DEP by only running trusted code. No code injection needed!

See the Voting machine story:

- Security Now 211

http://www.security-faqs.com/what-is-rop-and-how-do-hackers-use-it.html

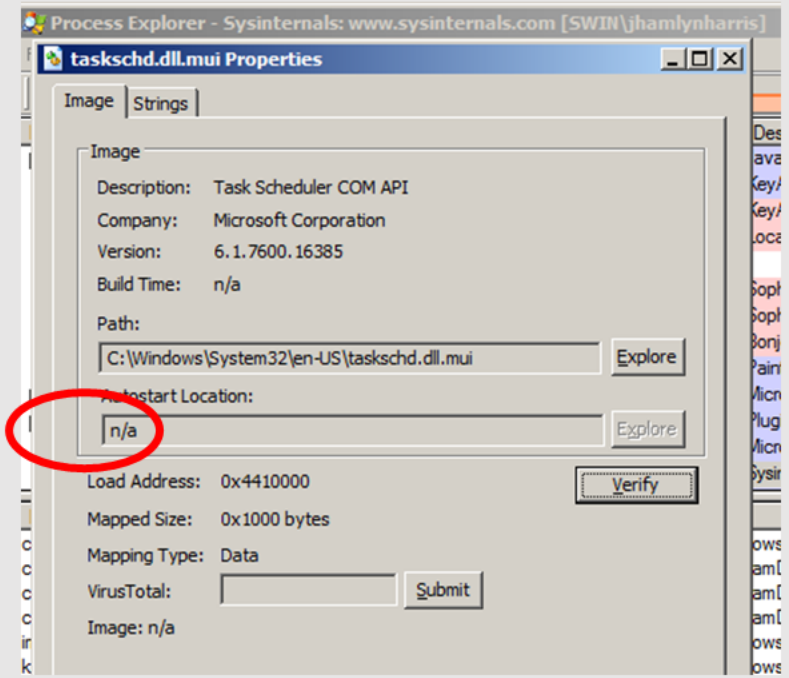
Data Execution Prevention

DEP is supported in Windows and in Linux kernels since XP SP2 and Kernel 2.6.8 respectively.

It can be turned off in software

Some dlls don't use it.

http://en.wikipedia.org/wiki/Data_Execution_Prevention



processes not using ASLR