Acknowledgement

- This lecture is extended and modified from lecture notes by:
  - Dr. Cliff Zou: CAP6135/CIS3360 courses
  - Dr. Yan Chen: EECS350 course
  - Dr. Nickolai Zeldovich: 6.858 course
  - Dr. Dang Song: CS161 course
  - Dr. Marco Cova 20009/20010 courses

Contents

- Stack Overflow Overview
- What causes buffer overflow?
- Stack Overflow Defense

- Stack Overflow Overview

The Problem

```
void foo(char *s) {
  char buf[10];
  strcpy(buf,s);
  printf("buf is %s\n",s);
}
…
foo("thisstringistolongforfoo");
```

Exploitation

- The general idea is to give servers very large strings that will overflow a buffer.
- For a server with sloppy code – it's easy to crash the server by overflowing a buffer.
- It's sometimes possible to actually make the server do whatever you want (instead of crashing).

Necessary Background

- C functions and the stack.
- A little knowledge of assembly/machine language.
- How system calls are made (at the level of machine code level).
- **`exec()`** system calls

  – How to "guess" some key parameters.

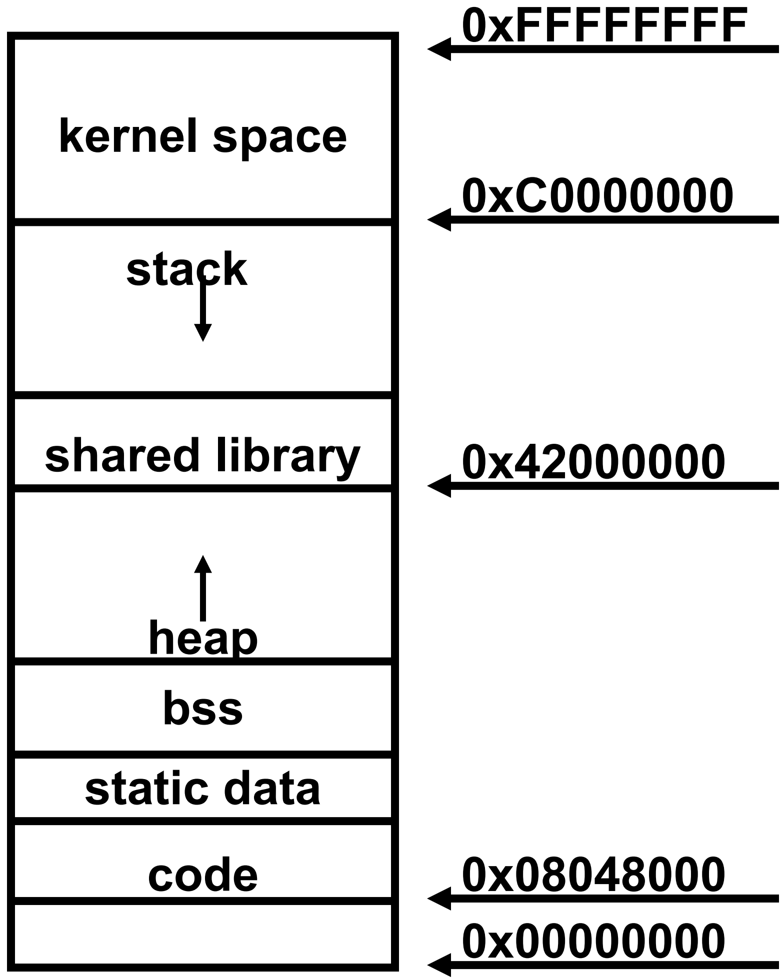
What is a Buffer Overflow?

- Intent
  - Arbitrary code execution
    - Spawn a remote shell or infect with worm/virus
  - Denial of service
    - Cause software to crash
      - E.g., ping of death attack
- Steps
  - Inject attack code into buffer
  - Overflow return address
  - Redirect control flow to attack code
  - Execute attack code

Attack Possibilities

- Targets
  - Stack, heap, static area
  - Parameter modification (non-pointer data)
    - Change parameters for existing call to exec()
    - Change privilege control variable
- Injected code vs. existing code
- Absolute vs. relative address dependence
- Related Attacks
  - Integer overflows
  - Format-string attacks

Address Space

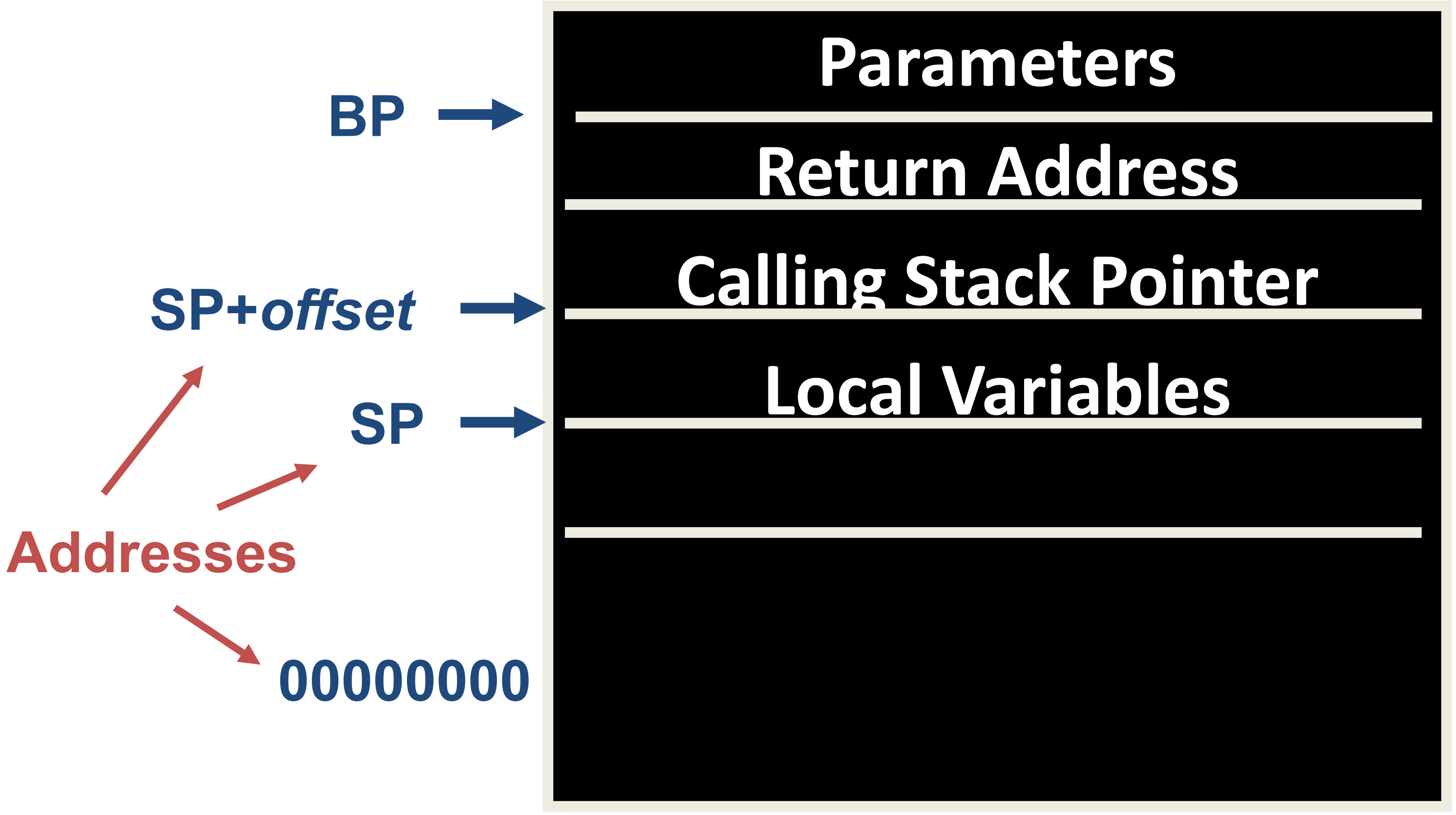


From Dawn Song's RISE: http://research.microsoft.com/projects/SWSecInstitute/slides/Song.ppt

C Call Stack

- C Call Stack
  - When a function call is made, the return address is put on the stack.
  - Often the values of parameters are put on the stack.
  - Usually the function saves the stack frame pointer (on the stack).
  - Local variables are on the stack.

A Stack Frame



SP: stack pointer   BP: base/frame pointer

Sample Stack

18

addressof(y=3) ***return address***

saved stack pointer

buf

y

x

```
x=2;
foo(18);
y=3;
```

```
void foo(int j) {
    int x,y;
    char buf[100];
    x=j;
    …
}
```

"Smashing the Stack"*

- The general idea is to overflow a buffer so that it overwrites the return address.
- When the function is done it will jump to whatever address is on the stack.
- We put some code in the buffer and set the return address to point to it!

*taken from the title of an article in Phrack 49-7

Before and After

```
void foo(char *s) {
     char buf[100];
     strcpy(buf,s);
     …
```

- What causes buffer overflow?

Example: gets()

```
char buf[20];
gets(buf);  // read user input until
            // first EoL or EoF character
```

- Never use **gets**
- Use **fgets(buf, size, stdout)** instead

Example: strcpy()

char dest[20];

strcpy(dest, src); // copies string src to dest

- **strcpy** assumes **dest** is long enough , and assumes **src** is null-terminated

- Use **strncpy(dest, src, size)** instead

Spot the defect! (1)

```
char buf[20];
char prefix[] = ”http://”;
...
strcpy(buf, prefix);
   // copies the string prefix to buf
strncat(buf, path, sizeof(buf));
   // concatenates path to the string buf
```

Spot the defect! (1)

```
char buf[20];
char prefix[] = ”http://”;
...
strcpy(buf, prefix);
   // copies the string prefix to buf
strncat(buf, path, sizeof(buf));
   // concatenates path to the string buf
```

strncat's 3rd parameter is number of chars to copy, not the buffer size

Another common mistake is giving **sizeof(path)** as 3rd argument...

Spot the defect! (2)

```
char src[9];
char dest[9];

char base_url = "www.ru.nl";
strncpy(src, base_url, 9);
   // copies base_url to src
strcpy(dest, src);
   // copies src to dest
```

base_url is 10 chars long, incl. its null terminator, so **src** won't be not null-terminated

so **strcpy** will overrun the buffer **dest**

Example: strcpy and strncpy

- Don't replace **strcpy(dest, src)** by
  - **strncpy(dest, src, sizeof(dest))**
- but by
  - **strncpy(dest, src, sizeof(dest)-1)**
  - **dest[sizeof(dest)-1] = `\0`;**
  - if **dest** should be null-terminated!

- A strongly typed programming language could of course enforce that strings are always null-terminated...

Spot the defect! (3)

```
char *buf;
int i, len;
read(fd, &len, sizeof(len));
buf = malloc(len);
read(fd,buf,len);
```

Spot the defect! (3)

```
char *buf;
int i, len;
read(fd, &len, sizeof(len));
buf = malloc(len);
read(fd,buf,len);
```

Didn’t check if negative

len cast to unsigned and negative length overflows

- **Memcpy() prototype:**
  - **void *memcpy(void *dest, const void *src, size_t n);**
- **Definition of size_t: typedef unsigned int size_t;**

Implicit Casting Bug

- **A signed/unsigned or an implicit casting bug**
  - **Very nasty – hard to spot**
- **C compiler doesn't warn about type mismatch between signed int and unsigned int**
  - **Silently inserts an implicit cast**

Spot the defect! (4)

May results in integer overflow

```
char *buf;
int i, len;
read(fd, &len, sizeof(len));
if (len < 0)
   {error ("negative length"); return; }
buf = malloc(len+5);
read(fd,buf,len);
buf[len] = '\0'; // null terminate buf
```

Spot the defect! (5)

```
#define MAX_BUF = 256

void BadCode (char* input)
   {     short len;
         char buf[MAX_BUF];
         len = strlen(input);
         if (len < MAX_BUF)
              strcpy(buf,input);
   }
```

What if input is longer than 32K ?
len will be a negative number,
due to integer overflow
hence: potential buffer overflow

Spot the defect! (6)

```
char buff1[MAX_SIZE], buff2[MAX_SIZE];
// make sure it’s a valid URL and will fit
if (! isValid(url)) return;
if (strlen(url) > MAX_SIZE – 1) return;
// copy url up to first separator, ie. first ’/’, to buff1
out = buff1;
do {
    // skip spaces
    if (*url != ’ ’) *out++ = *url;
} while (*url++ != ’/’);
strcpy(buff2, buff1);
...
```

what if there is no ‘/’ in the URL?

Loop termination (exploited by Blaster worm)

Spot the defect! (7)

```
#include <stdio.h>
int main(int argc, char* argv[])
{ if (argc > 1)
printf(argv[1]);
return 0;
}
```

This program is vulnerable to **format string** attacks, where calling the program with strings containing special characters can result in a buffer overflow attack.

Format String Attacks

- int printf(const char *format [, argument]…);
  - snprintf, wsprintf …
- What may happen if we execute

  printf(string);

  - Where **string** is user-supplied ?
  - If it contains special characters, eg %s, %x, %n, %hn?

Format String Attacks

- Why this could happen?
  - Many programs delay output message for batch display:
  - fprintf(STDOUT, err_msg);
  - Where the err_msg is composed based on user inputs
  - If a user can change err_msg freely, format string attack is possible

Format String Attacks

- %**x** reads and prints 4 bytes from stack
  - this may leak sensitive data
- %**n** writes the number of characters printed so far onto the stack
  - this allow stack overflow attacks...
- C format strings break the “don’t mix data & code” principle.
- “Easy” to spot & fix:
  - replace **printf(str) by printf(“%s”, str)**

Use Unix Machine in Department

- The Unix machine: eustis.eecs.ucf.edu
- Must use SSH to connect
  - Find free SSH clients on Internet
    - E.g., Putty (command line based)
    - http://en.wikipedia.org/wiki/Ssh_client
    - Find a GUI-based SSH client
- Username: NID
- Default password:
  the first initial of your last name in uppercase and the last 5 digits of your PID

Example of “%x” --- Memory leaking

```
#include <stdio.h>
void main(int argc, char **argv){
      int a1=1; int a2=2;
      int a3=3; int a4=4;
      printf(argv[1]);
}
```

czou@:~$ ./test
czou@eustis:~$ ./test "what is this?"
what is this?czou@eustis:~$
czou@eustis:~$
czou@eustis:~$ ./test "%x   %x   %x   %x   %x %x"
4   3   2   1   bfc994b0 bfc99508czou@eustis:~$
czou@eustis:~$

Bfc994b0:   saved stack pointer
Bfc99508:   return address

```
#include <stdio.h>
void foo(char *format){
    int a1=11; int a2=12;
    int a3=13; int a4=14;
    printf(format);
}
void main(int argc, char **argv){
    foo(argv[1]);
    printf("\n");
}
```

$./format-x-subfun "%x %x %x %x : %x,   %x,   %x   "

80495bc **e d c : b**,   bffff7e8,   **80483f4**

Four variables   Return address

- **What does this string ("%x:%x:%s") do?**
  - **Prints first two words of stack memory**
  - **Treats next stack memory word as memory addr and prints everything until first '\0'**
    - **Could segment fault if goes to other program's memory**

- **Use obscure format specifier (%n) to write any value to any address in the victim's memory**
  - **%n --- write 4 bytes at once**
  - **%hn --- write 2 bytes at once**
- **Enables attackers to mount malicious code injection attacks**
  - **Introduce code anywhere into victim's memory**
  - **Use format string bug to overwrite return address on stack (or a function pointer) with pointer to malicious code**

Example of “%n”---- write data in memory

```
#include <stdio.h>
void main(int argc, char **argv){
    int bytes;
    printf(“%s%n\n”, argv[1], &bytes);
    printf(“You input %d characters\n”, bytes);
}
```

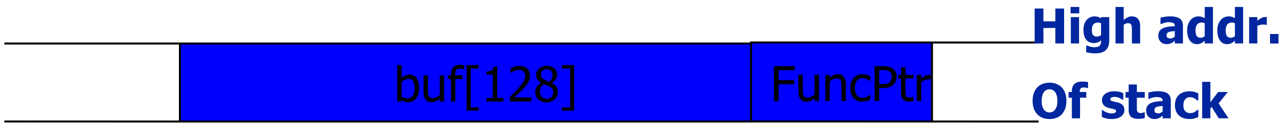
$./test hello

hello

You input 5 characters

Function Pointer Overwritten

- Function pointers: (used in attack on PHP 4.0.2)



  - Overflowing buf will override function pointer.
  - Harder to defend than return-address overflow attacks

Test by Yourself

```
#include <stdio.h>
void main(void){
  /* short x = 32767;*/
  unsigned short x = 65535;
   x = x +1;
   printf("x= %d\n", x);
}
```

- Try to run it to see how overflow happens.
  - Modify the x definition to see other integer overflow cases

Buffer Overflow Defense

Countermeasures

- We can take countermeasures at different points in time
  - before we even begin programming
  - during development
  - when testing
  - when executing code
- to prevent, to detect – at (pre)compile time or at runtime -, and to migitate problems with buffer overflows

Preventing Buffer Overflow Attacks

- Non-executable stack
- Static source code analysis.
- Run time checking: StackGuard, Libsafe, SafeC, (Purify).
- Randomization.
- Type safe languages (Java, ML).
- Detection deviation of program behavior
- Sandboxing
- Access control …

Prevention

- Don’t use C or C++ (use type-safe language)
  - Legacy code
  - Practical?
- Better programmer awareness & training
  - Building Secure Software, J. Viega & G. McGraw, 2002
  - Writing Secure Code, M. Howard & D. LeBlanc, 2002
  - 19 deadly sins of software security, M. Howard, D LeBlanc & J. Viega, 2005
  - Secure programming for Linux and UNIX HOWTO, D. Wheeler, www.dwheeler.com/secure-programs
  - Secure C coding, T. Sirainen www.irccrew.org/~cras/security/c-guide.html

Dangerous C system calls

source: Building secure software, J. Viega & G. McGraw, 2002

Extreme risk
- gets

High risk
- strcpy
- strcat
- sprintf
- scanf
- sscanf
- fscanf
- vfscanf
- vsscanf

High risk (cntd)
- streadd
- strecpy
- strtrns
- realpath
- syslog
- getenv
- getopt
- getopt_long
- getpass

Moderate risk
- getchar
- fgetc
- getc
- read
- bcopy

Low risk
- fgets
- memcpy
- snprintf
- strccpy
- strcadd
- strncpy
- strncat
- vsnprintf

Secure Coding

- **Avoid risky programming constructs**
  - **Use fgets instead of gets**
  - **Use strn* APIs instead of str* APIs**
  - **Use snprintf instead of sprintf and vsprintf**
  - **scanf & printf: use format strings**
- **Never assume anything about inputs**
  - **Negative value, big value**
  - **Very long strings**

Prevention – use better string libraries

- there is a choice between using statically vs dynamically allocated buffers
  - static approach easy to get wrong, and chopping user input may still have unwanted effects
  - dynamic approach susceptible to out-of-memory errors, and need for failing safely

Better string libraries

- **libsafe.h** provides safer, modified versions of eg strcpy
- **strlcpy**(dst,src,size) and **strlcat**(dst,src,size**)** with size the size of dst, not the maximum length copied.
  - Used in OpenBSD
- **glib.h** provides Gstring type for dynamically growing null-terminated strings in C
  - but failure to allocate will result in crash that cannot be intercepted, which may not be acceptable
- **Strsafe.h** by Microsoft guarantees null-termination and always takes destination size as argument
- **C++ string class**
  - data() and c-str()return low level C strings, ie char*, with result of data()is not always null-terminated on all platforms...

Dynamic countermeasures

- Protection by kernel
  - Non-executable stack memory (NOEXEC)
    - prevents attacker executing her code
  - Address space layout randomisation (ASLR)
    - generally makes attacker's life harder
      - E.g., harder to get return address place and injected code address
- Protection inserted by the compiler
  - to prevent or detect malicious changes to the stack
- Neither prevents against heap overflows

Bugs to Detect in Source Code Analysis

- Some examples

- Crash Causing Defects
- Null pointer dereference
- Use after free
- Double free
- Array indexing errors
- Mismatched array new/delete
- Potential stack overrun
- Potential heap overrun
- Return pointers to local variables
- Logically inconsistent code

- Uninitialized variables
- Invalid use of negative values
- Passing large parameters by value
- Underallocations of dynamic data
- Memory leaks
- File handle leaks
- Network resource leaks
- Unused values
- Unhandled return codes
- Use of invalid iterators

Marking stack as non-execute

- **Basic stack exploit can be prevented by marking stack segment as non-executable or randomizing stack location.**
  - **Then injected code on stack cannot run**
  - **Code patches exist for Linux and Solaris**
    - **E.g., our olympus.eecs.ucf.edu has patched for stack radnomization**
- **Problems:**
  - **Does not block more general overflow exploits:**
    - **Overflow on heap, overflow func pointer**
  - **Does not defend against `return-to-libc' exploit.**
  - **Some apps need executable stack (e.g. LISP interpreters).**

Randomization Techniques

- **For successful exploit, the attacker needs to know where to jump to, i.e.,**
  - **Stack layout for stack smashing attacks**
  - **Heap layout for code injection in heap**
  - **Shared library entry points for exploits using shared library**
- ***R*andom*iz*ation Techniques for Software *Se*curity**
  - **Randomize system internal details**
    - **Memory layout**
    - **Internal interfaces**
  - **Improve software system security**
    - **Reduce attacker knowledge of system detail to thwart exploit**
    - **Level of indirection as access control**

Randomize Memory Layout (I)

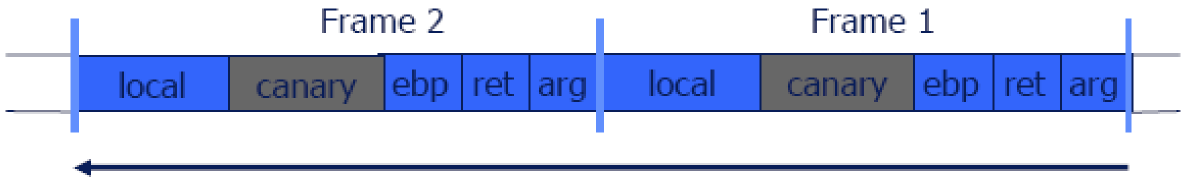
- **Randomize stack starting point**
  - **Modify execve() system call in Linux kernel**
  - **Similar techniques apply to randomize heap starting point**
- **Randomize heap starting point**
- **Randomize variable layout**

Randomize Memory Layout (II)

- **Handle a variety of memory safety vulnerabilities**
  - **Buffer overruns**
  - **Format string vulnerabilities**
  - **Integer overflow**
  - **Double free**
- **Simple & Efficient**
  - **Extremely low performance overhead**
- **Problems**
  - **Attacks can still happen**
    - **Overwrite data**
    - **May crash the program**
  - **Attacks may learn the randomization secret**
    - **Format string attacks**

Dynamic countermeasure: stackGuard

- **Solution: StackGuard**
  - **Run time tests for stack integrity.**
  - **Embed "canaries" in stack frames and verify their integrity prior to function return.**



Canary Types

- **Random canary:**
  - **Choose random string at program startup.**
  - **Insert canary string into every stack frame.**
  - **Verify canary before returning from function.**
  - **To corrupt random canary, attacker must learn the random string.**

Canary Types

- Additional countermeasures:
  - use a random value for the canary
  - XOR this random value with the return address
  - include string termination characters in the canary value (why?)

- **StackGuard implemented as a GCC patch**
  - **Program must be recompiled**
- **Low performance effects: 8% foR Apache**
- **Problem**
  - **Only protect stack activation record (return address, saved ebp value)**

Purify

- **A tool that developers and testers use to find memory leaks and access errors.**
- **Detects the following at the point of occurrence:**
  - **reads or writes to freed memory.**
  - **reads or writes beyond an array boundary.**
  - **reads from uninitialized memory.**

Purify - Catching Array Bounds Violations

- **To catch array bounds violations, Purify allocates a small "red-zone" at the beginning and end of each block returned by malloc.**
- **The bytes in the red-zone → recorded as unallocated.**
- **If a program accesses these bytes, Purify signals an array bounds error.**
- **Problem:**
  - **Does not check things on the stack**
  - **Extremely expensive**

Further improvements

- PointGuard
  - also protects other data values, eg function pointers, with canaries
    - Higher performance impact than stackGuard
- ProPolice's Stack Smashing Protection (SSP) by IBM
  - also re-orders stack elements to reduce potential for trouble
- Stackshield has a special stack for return addresses, and can disallow function pointers to the data segment

Dynamic countermeasures

- libsafe library prevents buffer overruns beyond current stack frame in the dangerous functions it redefines
  - Dynamically loaded library.
  - Intercepts calls to strcpy (dest, src)
    - Validates sufficient space in current stack frame:
      |frame-pointer – dest| > strlen(src)
    - If so, does strcpy.
      Otherwise, terminates application.

Dynamic countermeasures

- libverify enhancement of libsafe keeps copies of the stack return address on the heap, and checks if these match

- None of these protections are perfect!
  - even if attacks to return addresses are caught, integrity of other data other than the stack can still be abused
  - clever attacks may leave canaries intact
  - where do you store the "master" canary value
    - a cleverer attack could change it
  - none of this protects against heap overflows
    - eg buffer overflow within a struct...
  - New proposed non-control attack

Summary

- Buffer overflows are the top security vulnerability
- Any C(++) code acting on untrusted input is at risk
- Getting rid of buffer overflow weaknesses in C(++) code is hard (and may prove to be impossible)
  - Ongoing arms race between countermeasures and ever more clever attacks.
  - Attacks are not only getting cleverer, using them is getting easier