

A brief history of C

- The Unix operating system grew out of work carried out at Bell Laboratories during the late 1960s and early 1970s
- It was originally implemented in assembly on a PDP-7 by Ken Thompson and Dennis Ritchie
- Based on this experience Ritchie decided to develop a new system language, one that could be used to implement Unix and thereby facilitate its porting to other architectures
- C was developed to meet this need and although the Unix kernel for the PDP-11 was mostly coded in assembly, in 1973 the entire kernel was rewritten in C



A brief history of C (continued)

- Unix was one of the first operating system kernels not to be coded in assembly
- Given its origins, C's design goals are logical: it had to be portable, provide for low level memory access and require minimal runtime support
- In 1978 the first edition of K&R's "The C Programming Language" appeared and provided an informal specification of the language for several years
- However, with different interpretations and versions appearing, standardisation became an issue and in 1983 ANSI formed a committee to establish the C standard



A brief history of C (continued)

- The committee produced the ANSI C standard, ratified in 1989 and described in the second edition of K&R's "The C Programming Language"
- This is Standard C, ANSI C or C89
- In 1990 the ANSI C standard was accepted with some minor modifications by ISO giving C90
- C underwent further development in the 1990s and in 1999 ISO published a new standard, C99 which was subsequently adopted by ANSI



A brief history of C (continued)

- Despite its age and roots as a system language, C is today widely employed as a general purpose application development language
- Given its efficiency and performance, C is particularly well suited to implementing libraries and interpreters of other high level languages



How popular is C?

- C seems to be on the decline in terms of “popularity” although it still ranks highly...
- (Source <http://www.tiobe.com>)

Sep 2007	Sep 2006	Language	Rating	Delta
1	1	Java	21.701%	+0.17%
2	2	C	14.908%	-3.15%
3	4	VB	10.748%	+0.12%
4	5	PHP	10.284%	+1.08%
5	3	C++	9.938%	-0.82%

- So what's the problem with C?



Strong and weak typing

- C is regarded as a weakly typed language, many type conversions are implicit and need not be implemented by the programmer
- On the other hand Java is traditionally seen as a strongly typed language, conversions being left to the programmer to carry out through type casting



Memory safety

- The concept of memory safety is one closely related to that of type safety
- For a programming language to be memory safe, writes to a given type must not exceed the bounds of that type
- Again, Java is memory safe, C is not



Where are we?

- We have built up an understanding of both process organisation in memory and in particular the role of the stack in program execution
- As such, we are ready to take a first look at some of the more common vulnerabilities introduced through C coding error
- We do not yet investigate in detail how these vulnerabilities can be exploited: we limit ourselves to understanding them and outlining how, in principle, they could be exploited
- Exploit development and implementation is covered in detail in upcoming lectures



- Notice that the size of the `dest` array is not passed as a parameter to `strcpy`
- Obviously, the size of `dest` is in no way taken into account in the implementation of `strcpy`
- `strcpy`, being oblivious to the size of `dest`, will copy from `src` until it reaches the end of string character: `'\0'`
- When `src` is bigger than `dest` and once `dest` has been filled, `strcpy` will continue to write beyond its bounds and `dest` overflows: we have a classic buffer overflow



- The overflow must go somewhere, where does it go?
- `strcpy` writes from low to high memory addresses and so will start at the “bottom” of `dest` and work its way up the stack overwriting whatever lies along the way
- Crucially, not far above `dest` on the stack lies the stored return address: it can be overwritten during an overflow
- Not only can the return address be overwritten but it can be purposefully rewritten to point to a new location



- Where will the new return address point to? Usually somewhere inside `src` itself where the attacker has placed executable code
- Injected attack code (the “payload”) is usually padded with `NOOP` instructions to widen the landing area for a guessed return address
- A padded landing area may also help circumvent defensive techniques such as stack randomisation
- Sometimes the code which an attacker aims to execute may already exist in the process address space and need not be injected at all (or we may seek to skip a block of privilege-checking code)



- Traditional buffer overflows are stack-based but they need not be
- In the following C excerpt `strcpy` is used to write to a BSS-based character array
- Above the array, in the BSS section of the process image, lies a function pointer
- By overflowing the array an attacker can reach the function pointer and rewrite it to point at some other function



```

char dest[64];
void (*fptr)(void);

void
foo(char *src)
{
    fptr = bar;

    strcpy(dest, src);

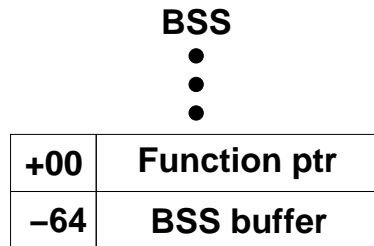
    (*fptr)();
}

```

```

void
bar(void)
{
    printf("`Hi\n'");
}

```



Off-by-one errors

- An off-by-one error allows an attacker to overflow a buffer by one byte and in so doing to possibly overwrite the least significant byte of the calling function's EBP
- If the attacker is lucky the calling function's now modified EBP may point into an area under her control e.g. the overflowing buffer itself
- The attacker might try filling the buffer with the address of some code she would like to execute
- What happens when the overflow occurs?

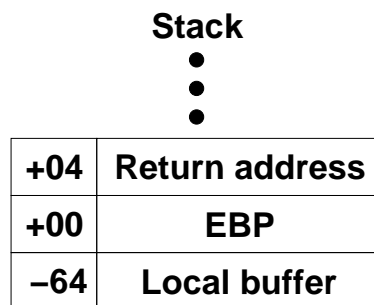
```

void
foo(char *src)
{
    char dest[64];

    strncpy(dest,
            src,
            sizeof (dest));

    dest[sizeof (dest)] =
        '\0';
}

```



- Disassembling `foo`, it ends with:


```
leave [mov ebp, esp; pop ebp]
ret [pop eip]
```
- Similarly the calling function ends with:


```
leave [mov ebp, esp; pop ebp]
ret [pop eip]
```
- On leaving the `foo` function the calling function's now corrupted EBP is popped and execution continues in the calling function
- On leaving the calling function the stack pointer ESP is set to the corrupted EBP (`mov ebp, esp`)

