

# The C Programming Language

A workshop about C programming basics  
and security issues

# Programming basics

- Syntax
  - Statements and separation thereof
  - Conditionals, Loops
  - Functions and return values
  - Pointers
- Compiled language
- Memory layout

# Statements and separation

- Separation

a; b; c; (Semicolon is separator)

- Assignment

a = b + c;

a = b = c;

Not: a(b) = c; (lvalue is read-only)

- Escaping

char \*str = "What is a \"hacker\"?";

# Conditionals and loops

- Simple conditional

```
if (a == b && (c > d || e != f))  
    g = do_something(a);
```

- Multiple choice

```
switch (e) { case 23: asdf(); case 42: bsdf();  
            break; default: csdf(); }
```

- Iteration

```
for (i = 1; i < argc; i++) puts(argv[i]);
```

- Conditional loop

```
while (i < 10) i++;
```

# Functions and return values

- Declaration

```
int add(int i, int j) { int k = 23; return i + j *  
    k; }
```

- Calling

```
int x = add(a,b);
```

- Pointers

```
struct math_funcs *funcs = { .add = add; }
```

```
int x = funcs->add(a, b);
```

- Object oriented programming!

# Functions and return values

- Write small functions!
- Every subtask should be its own functions
- Use function scopes! (static, extern, etc.)
  - static: function visible within the same object  
(== compiled C source file)
  - extern: functions imported from a different object
  - inline: function may be integrated into other functions by the compiler in order to avoid function call overhead

# Pointers

- Pointers are addresses
- Pointers are incremental numbers of bytes in the memory
- A pointer really is a pointer, no matter which type

```
unsigned long str = htonl(('H' << 24) |  
    ('i' << 16));  
puts((char *) &str);
```

- Pointers increase by the size of their type
- `void *` is untyped, `sizeof(void)` a GNUism

# Pointers

- Pointer arithmetic is efficient

```
struct asdf *ptr = &asdf_str;
```

```
while (*ptr != NULL)
```

```
{
```

```
    do_something(ptr);
```

```
    ptr++; # <- ptr++ is an increment
```

```
}
```

```
struct asdf arr[SOME_LENGTH];
```

```
for (int i = 0; i < SOME_LENGTH; i++)
```

```
    do_something(arr[i]); # <- arr[i] is a multiplication
```

# Pointers

- Pointers are even more efficient

```
void do_something_struct(struct asdf a)
{
    // Help, I get a copy of the entire 168-byte struct!
}

void do_something_ptr(struct asdf *a)
{
    // Relief! I only get a copy of the 8-byte pointer!
}
```

# Pointers

- Pointers are helpful

```
char *str = "Don't let the sun go down on you.";
```

```
char *ptr = str;
```

```
while (*ptr != 'u' && *ptr != NULL)
```

```
    ptr++;
```

```
if (*ptr == 'u')
```

```
    *ptr = 'a'
```

```
puts(ptr);
```

- `strfry` and `memfrob` are not only pointless, they are GNU extensions

# Pointers

- Pointer casts are versatile

```
unsigned long str = ntohl(1214840832); // MSB number!  
puts((unsigned char *) &str);
```

- A pointer is simply a memory address
- Types are a «compiler invention»
- Pointers are pure mathematics
- Pure mathematics are cool
  - q.e.d.

# Using the compiler

- Warnings should always be considered fatal
- Enable all warnings
  - gcc -W -Wall
- Patch your gcc to include additional warnings (e.g. from the NetBSD gcc)
- Don't ignore warnings about unused return values!
  - <http://mysql.is-broken.ch/>
- Don't trust gcc 4.x for  $x < 3$

# Using the compiler

- Create shared libraries for much of your code!
  - Shared libraries only get loaded once if your program is run twice
  - Shared libraries allow you to reuse code in other programs
- `gcc -fPIC -c file.c`
- `gcc -o libblah.so -shared *.o`
- Don't forget the `-fPIC` or the amd64 guys will get you

# Memory layout

- 3 layers of memory contents:
  - Executed code (text)
  - Function-local memory (stack)
  - Global storage memory (heap)
- Example alignment (machine dependent)

Stack
empty
Heap
Text

# Memory layout

- Function's memory

```
int func(int i, int j)
```

```
{
```

```
    int k;
```

```
    unsigned char text[23];
```

```
    ...
```

text	23 byte
Gap	1 byte
k	4 byte
j	4 byte
i	4 byte
Saved stack pointer	8 byte

# Memory layout

- **Buffer overflow**

text	23 byte
Gap	1 byte
k	4 byte
j	4 byte
i	4 byte
Saved stack pointer	8 byte

- Write 44 bytes into buffer text (unchecked write)
  - Overwrites text, gap, k, j, i and the stack pointer
  - When execution of the function is done, jumps to the newly written address in the SP
  - New address can happen to be in some shell code which is stored in the variable text

# Examples from the books

```
void getanswer(void)
{
    char answer[256];
    puts("Type something:");
    gets(answer);
    printf("You typed \"%s\"\n", answer);
}
```

- What's wrong here?
  - perl -e'print("A" x 264);' | program

# Examples from the books

```
void getanswer(void)
{
    char answer[256];
    puts("Type something:");
    fgets(answer, sizeof(answer) - 1, stdin);
    printf(answer);
}
```

- What's wrong here?
  - echo "%s%s%s%s" | program

# Examples from the books

- Imagine doing this over network sockets!
- MySQL user name buffer overflow ;-)

# Examples from the books

```
void *buf;
void sighandler(int sig)
{
    if (buf) free(buf);
    exit(-sig);
}
int main(void)
{
    signal(SIGPIPE, sighandler);
    signal(SIGSEGV, sighandler);
    ...
}
```

# Examples from the book

- What's wrong here?
  - kill -PIPE process
    - Frees buf
  - kill -SEGV process
    - double free
- Double free can lead to code execution under Linux! (But not under NetBSD)
- Also, don't call non-reentrant functions in signal handlers!
  - <http://emacs.is-broken.ch/>

# Cool documentation

- man functionname
  - Watch out for signs of GNU extensions!
- POSIX standard
  - <http://www.opengroup.org/onlinepubs/009695399/>
- Single UNIX Specification
  - <http://opengroup.org/onlinepubs/007908799/index.html>
- If it's not POSIX or SUS, don't use it!
  - Use other libs, however ;-)

eof

Now let's do some coding!

# And don't forget

It's fun to look at other people's code!