



Defensive Programming

Ulrich Drepper
Consulting Engineer



Security from Day 1

Applications the target now that the OS is hardened

Security must be no afterthought

- Programmers must be knowledgeable about problems and solutions
 - Education needed
 - Enforcement of guidelines
 - Testing of compliance
- Automatic security features can help
 - Programming languages with fewer pitfalls preferred
 - Use safer alternatives to interfaces which are problematic
 - Enable automatic fortification of code



Vulnerability Classes

Every programming language has problems in one or more of the following classes:

- Memory handling
- Filesystem use
- Untrusted input
- Unchecked resource usage

Some problems are inherent to the OS environment

Others are inherent to the programming language: choice means compromise



Memory Handling Problems

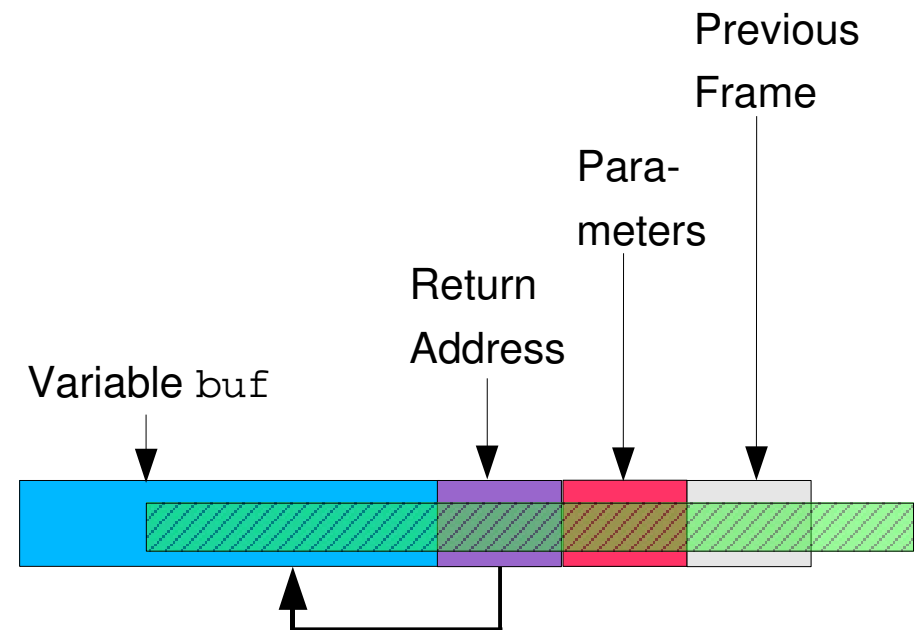
C and partially C++ vulnerable

Programmer is responsible for

- Memory allocation
- Memory deallocation
- Memory initialization

Buffer overflows are the biggest enemy

```
void hello(char *s) {  
    char buf[50];  
    strcpy(stpcpy(buf, "Hello "), s);  
    puts(s);  
}
```



Implicit Allocation

The best solution is to avoid explicit memory allocation altogether:

- Constructors in C++, Java, LISP, etc
 - Also assures memory is initialized
- Use alternative interfaces:
 - `asprintf()` to create simple strings
 - `open_memstream()` for more complicated strings
 - Use 'a' modifier in `scanf()`
 - `getline()` instead of `fgets()`
 - `strdup()` for string duplication



Automatic Protection

gcc and glibc have been extended to automatically protect several functions if possible:

- gcc keeps track of array and variable sizes
 - Variable sizes always known
 - Implicit and explicit `alloca()` tracked inside function
 - Memory allocated with `malloc()`-like functions tracked inside function

- Functions like `strcpy()` have wrapper macros or inline functions:

```
char *__strcpy_chk (char *dest, const char *src, size_t n)
```

- The `n` parameter describes the real length of the `dest` buffer
- The function checks the source string length before copying and fails if necessary



Prevented Overflow

```
#include <string.h>
int main(int argc, char *argv[]) {
    char buf[10];
    strcpy(buf, argv[1]);
    return 0;
}
```

Running this produces:

```
***buffer overflow detected ***: ./prg terminated
===== Backtrace: =====
/lib/libc.so.6(__chk_fail+0x41)[0xb1ecc5]
/lib/libc.so.6(__strcpy_chk+0x3d)[0xb1e355]
./prg[0x80483bd]
/lib/libc.so.6(__libc_start_main+0xc6)[0xa55de6]
./prg[0x804831d]
===== Memory map: =====
005b6000-005bf000 r-xp 00000000 03:08 288623      /lib/libgcc_s-4.0.0-20050405.so
...
```



Correct Filesystem Use

- File name is not file content
 - Relationship can change at any time
- File creation possible source of data loss
 - Ensure no file is overwritten
 - Multiple creators can interfere with each other
- File replacing can lead to missing data
 - At any one time a file with the given name must exist
- Also applies to directories

Solution: work with file descriptors instead of file names



Untrusted Input

Exploit: carefully crafted, malicious input processed by buggy program

Solution: Verify all input

- Verify input text to recognize invalid text:
 - Invalid character encoding
 - Cross Site Scripting
 - SQL injection
- For Unix domain sockets: verify source process and user/group ID
- Check origin of signals
- Use appropriate random data (pseudo RNG, true randomness)



Enforce Good Programming

- Use warnings the compiler can emit:
 - Add `-Wall -Wextra` to compiler command line
 - For C++ additionally use `-Wefc++`
 - Add `-Werror` to enforce no warnings during compilation
- Annotate source code
 - Mark obsolete interfaces with `deprecated` function attribute
 - Make sure return value is used with `warn_unused_result` attribute
 - Check for invalid NULL pointer parameters with `nonnull` attribute
- Other tools like splint might prove usable as well



Debugging Techniques

- Obviously, general debugger available
- Most problematic in C/C++: memory handling problems
 - Runtime tests
 - Simple checks in glibc (MALLOC_CHECK_, mtrace, mcheck)
 - malloc replacements: dmalloc, ElectricFence
 - Special compilation mode: mudflap
 - Debug mode in C++ runtime library
 - External program: valgrind
- Big debugging problem: repeating often hard or impossible
 - Getting information at the time of the failure: backtrace



mudflap

mudflap has several advantages over the other techniques:

- Catches several more types of bugs
- Much faster than valgrind; could be used in production
- If all files are compiled appropriately, all errors are discovered

```
int a[10], b[10];  
int main(void) { return a[11]; }
```

```
mudflap violation 1 (check/read): time...
```

```
pc=0xd94342 location=`test.c:2 (main)'
```

```
  /usr/lib/libmudflap.so.0(__mf_check+0x44) [0xd94342]
```

```
  ./test.c(main+0x53) [0x80486f3]
```

```
  /usr/lib/libmudflap.so.0(__wrap_main+0x1d8) [0xd9506e]
```

```
Nearby object 1: checked region begins 0B into and ends 8B after  
mudflap object 0x95361e8: name=`test.c:1 a'
```



Questions?

Comments?

Contact: drepper@redhat.com

Paper: <http://people.redhat.com/drepper/defprogramming.pdf>

