OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

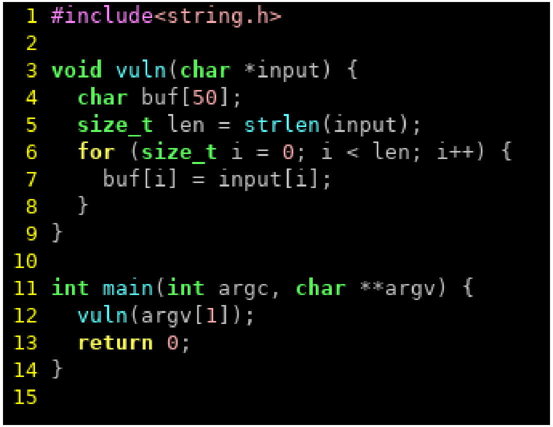
Overview Over Attack Vectors and Countermeasures for Buffer Overflows

Valentin Brandl
Wissenschaftliches Seminar
Faculty of Computer Science and Mathematics
OTH Regensburg

Topics

- Technical Overview
- Ways of Exploiting Buffer Overflows
- Analyzed Countermeasures
- Discussion

Technical Overview



```c
#include<string.h>

void vuln(char *input) {
  char buf[50];
  size_t len = strlen(input);
  for (size_t i = 0; i < len; i++) {
    buf[i] = input[i];
  }
}

int main(int argc, char **argv) {
  vuln(argv[1]);
  return 0;
}

```

argc	0xFE	← SP (main)
argv	0xFF	← BP (main)

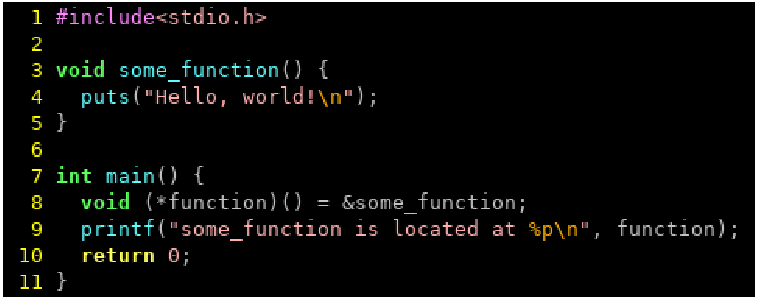
buf	0xC8	← SP (vuln)
buf	...	
buf	0xFA	← BP (vuln)
[old IP]	0xFB	
[BP (main)]	0xFC	
[*input]	0xFD	
argc	0xFE	
argv	0xFF	

[payload]	0xC8	← SP (vuln)
[payload]	...	
[payload]	0xFA	← BP (vuln)
[controlled IP]	0xFB	
[BP (main)]	0xFC	
[*input]	0xFD	
argc	0xFE	
argv	0xFF	

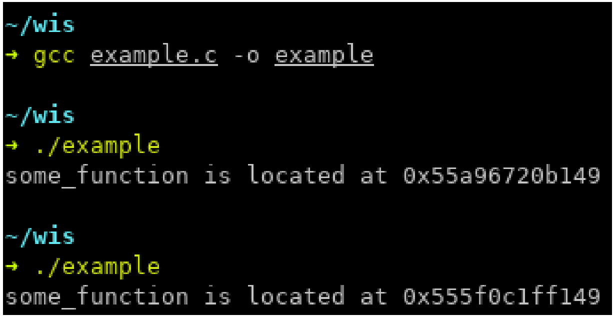
ASLR

- Randomize location of program in memory
- Attacker doesn‘t know where to jump
- Prevents code execution
- Does not prevent DoS
- Compile-time mitigation, no code changes needed

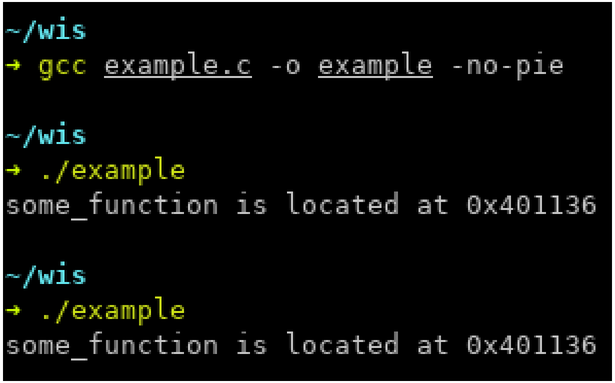
ASLR



```c
#include<stdio.h>

void some_function() {
  puts("Hello, world!\n");
}

int main() {
  void (*function)() = &some_function;
  printf("some_function is located at %p\n", function);
  return 0;
}
```



```
~/wis
➜ gcc example.c -o example

~/wis
➜ ./example
some_function is located at 0x55a96720b149

~/wis
➜ ./example
some_function is located at 0x555f0c1ff149
```



```
~/wis
➜ gcc example.c -o example -no-pie

~/wis
➜ ./example
some_function is located at 0x401136

~/wis
➜ ./example
some_function is located at 0x401136
```

NX

- Memory can be either writable or executable
- Attacker cannot supply shellcode directly
- Code reuse still possible
- Compile-time mitigation, no code changes needed

Stack Canary

- Markers at the end of a stack frame
- Invalid marker → Buffer overflow occurred
- No code changes required
- Only mitigates stack-based BOF
- Knowledge of canary allows bypassing

RAD

- Read-only stack for return addresses
- Compared before return
- Compiler extension
- Only against stack-based BOF

Bounds Checking

- Each indexing operation is checked
- 100% effective (where applied)
- Non-trivial runtime overhead

Dependent Types

- Value (size) is associated with a buffer
- Only allow indexing with validated values
- Language extension
- Lot of work to use, but type inference helps

State of the Art

- Major OS implement ASLR
- Compilers implement PIE, NX, Stack Canaries (discussable defaults)

Mitigation	GCC?	clang?
PIE	No	No
NX	Yes	Yes
Stack Canary	No	No

State of the Art

- Most techniques only prevent exploitation (code execution)
- DoS might be just as critical
- Only dependent typing and RBC actually prevent BOF

Conclusion

- Use C, C++ and Fortran only if unavoidable
- Viable alternatives exist (Rust, Go, Java, ...)

References

- Use C, C++ and Fortran only if unavoidable
- Viable alternatives exist (Rust, Go, Java, ...)