Buffer overflows

Buffer overflows

- Many CERT security incidents are still buffer overflows
- C and C++ do not perform array bound checks
- Possible to write past the end of an array and
  - Plant malicious code (payload)
  - Highjack control (by overwriting return address)

Buffer overflows

- Call stacks
- Vulnerable functions
- Shellcode
- Stack smashing
- Protection
  - Coding practice
  - Tool support (static, dynamic, and hybrid checks)

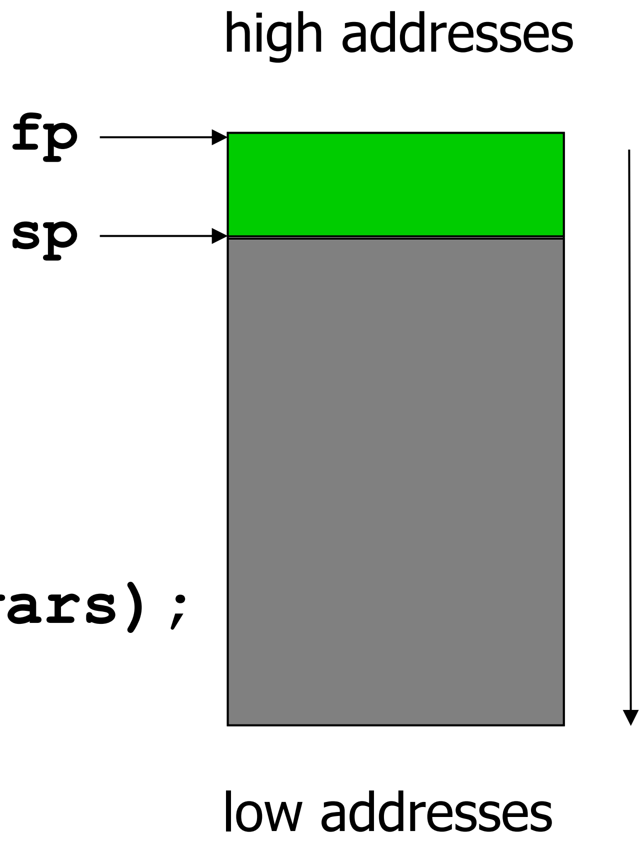
Call stacks

- Caller runs

```
push arg1;…; push argN;
push return_address;
```

- Callee runs

```
push fp;
fp := sp;
sp := sp + sizeof(local vars);
// body of callee
sp := fp;
fp := pop();
pc := pop();
```



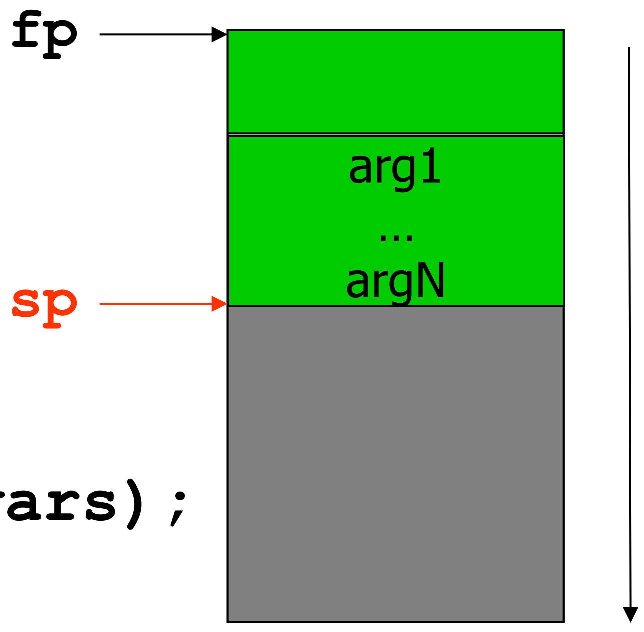
Call stacks

- Caller runs

```
push arg1;…; push argN;
push return_address;
```

- Callee runs

```
push fp;
fp := sp;
sp := sp + sizeof(local vars);
// body of callee
sp := fp;
fp := pop();
pc := pop();
```



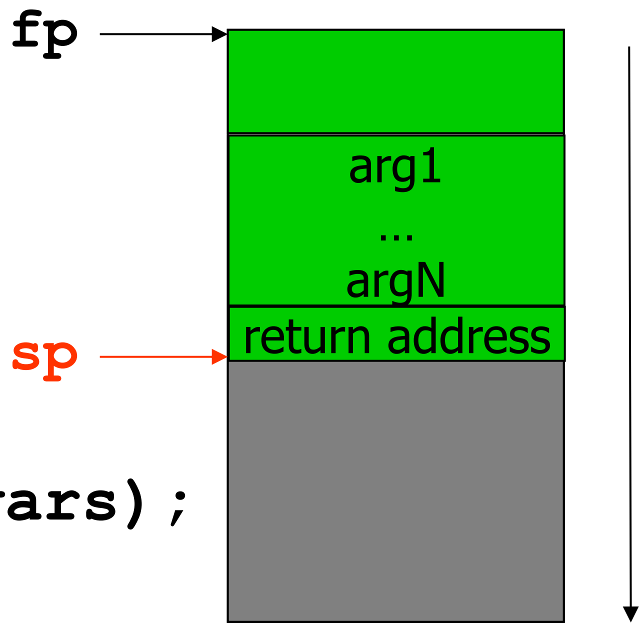
Call stacks

- Caller runs

```
push arg1;…; push argN;
push return_address;
```

- Callee runs

```
push fp;
fp := sp;
sp := sp + sizeof(local vars);
// body of callee
sp := fp;
fp := pop();
pc := pop();
```



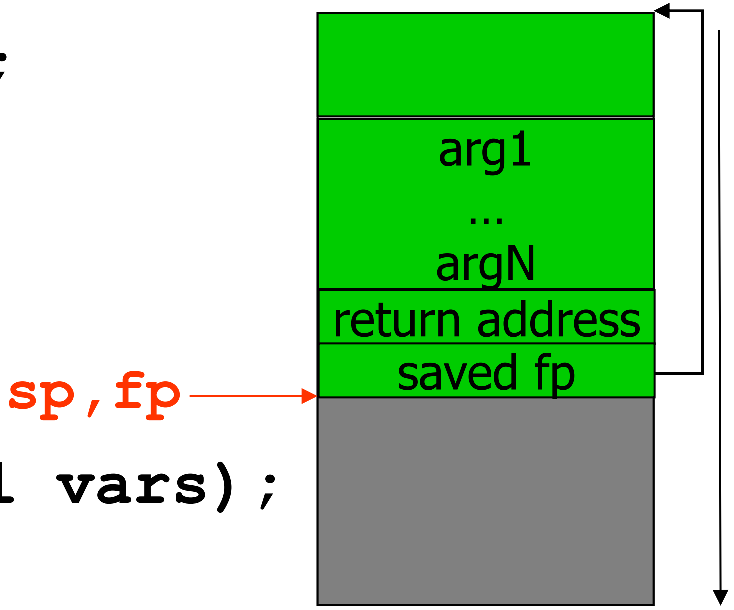
Call stacks

- Caller runs

```
push arg1;…; push argN;
push return_address;
```

- Callee runs

```
push fp;
fp := sp;
sp := sp + sizeof(local vars);
// body of callee
sp := fp;
fp := pop();
pc := pop();
```



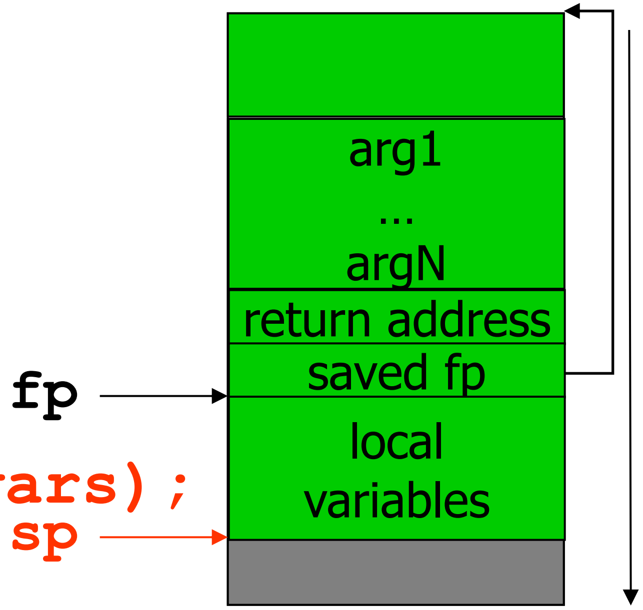
Call stacks

- Caller runs

```
push arg1;…; push argN;
push return_address;
```

- Callee runs

```
push fp;
fp := sp;
sp := sp + sizeof(local vars);
// body of callee
sp := fp;
fp := pop();
pc := pop();
```



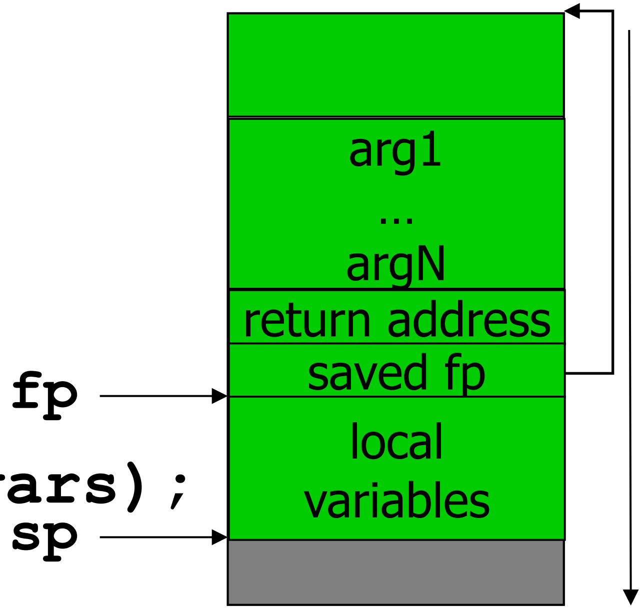
Call stacks

- Caller runs

```
push arg1;…; push argN;
push return_address;
```

- Callee runs

```
push fp;
fp := sp;
sp := sp + sizeof(local vars);
// body of callee
sp := fp;
fp := pop();
pc := pop();
```



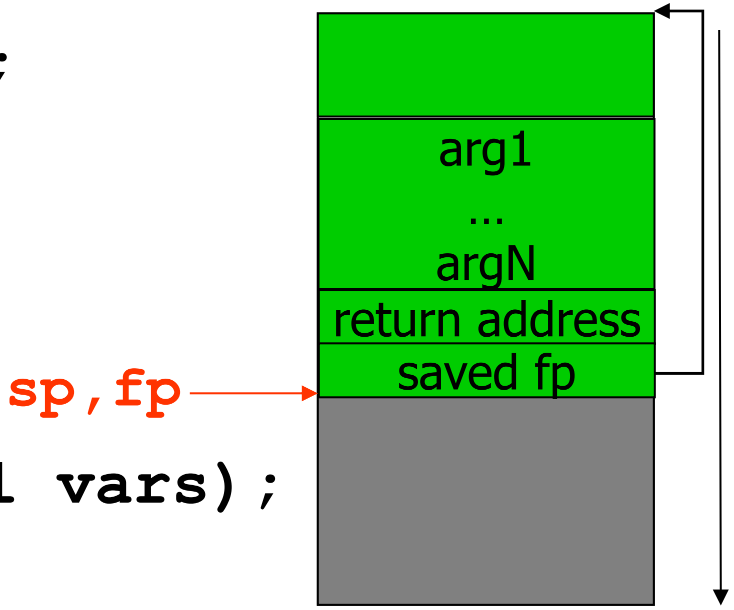
Call stacks

- Caller runs

```
push arg1;…; push argN;
push return_address;
```

- Callee runs

```
push fp;
fp := sp;
sp := sp + sizeof(local vars);
// body of callee
sp := fp;
fp := pop();
pc := pop();
```



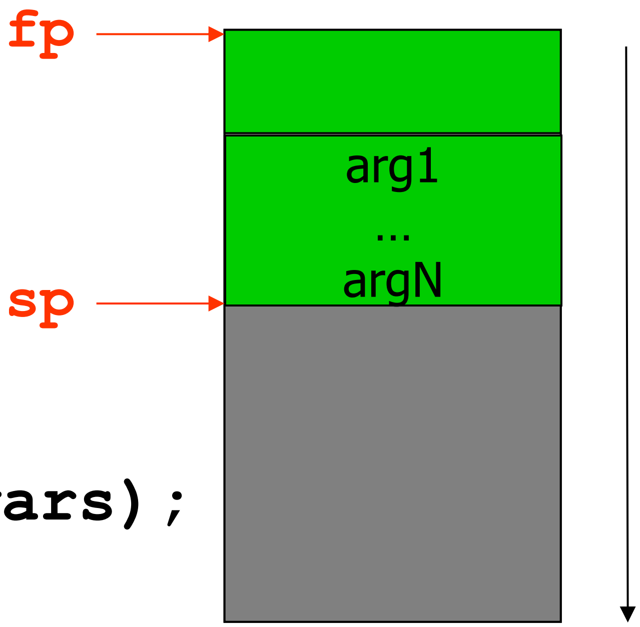
Call stacks

- Caller runs

```
push arg1;…; push argN;
push return_address;
```

- Callee runs

```
push fp;
fp := sp;
sp := sp + sizeof(local vars);
// body of callee
sp := fp;
fp := pop();
pc := pop();
```



Vulnerable functions

- strcpy, strcat, sprintf, scanf, sscanf, gets, read,…
- No bounds are checked
- Example: gets
  - Reads a buffer from stdin into the buffer
  - No checks for buffer overflows
  - \n (new line) or ^D (EOF) terminate the string

Shellcode

- Shellcode spawns a shell under the uid of the process
  - If uid is elevated to root, this will give rootshell
- How to make sure buffer address overwrites return address?
  - [shellcode][ADDR][ADDR][ADDR]…
- How to make sure return pointer will point to shellcode?
  - No-op sled: [NOP][NOP][NOP]…[shellcode]

Shellcode

- Use gcc and gdb to extract assembly and hex representations
- NOP on the x86 has the machine code 0x90
- How do you guess the ADDR to put in the payload?
  - find out where the first stack frame is: (deterministic in some Linux kernels)
  - offset can be calculated from experiments with different length overruns in gdb
- Attack string example:
  - [NOP][NOP][NOP]…[shellcode][ADDR][ADDR][ADDR]…

Stack smashing

- Vulnerable program:

```
char buf[100];
  …
gets(buf);
```

- Let us illustrate putting payload instead of the buffer

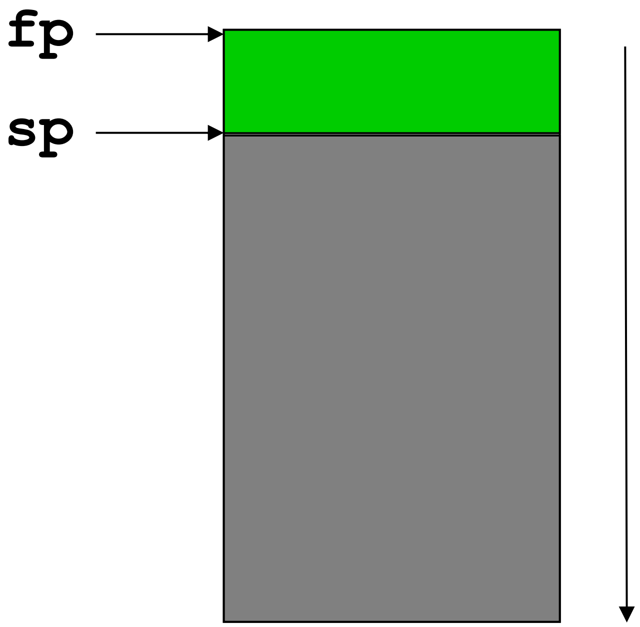
Stack smashing

- Caller runs

```
push return_address;
```

- Callee runs

```
push fp;
fp := sp;
// allocate space for buffer
sp := sp + sizeof(buffer);
gets(buffer);
// user enters shellcode
// gets returns
sp := fp;
fp := pop();
pc := pop();
```



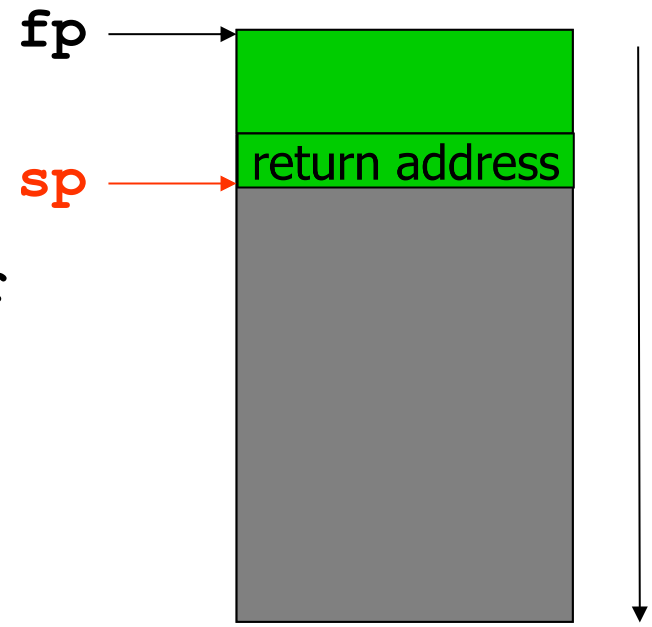
Stack smashing

- Caller runs

```
push return_address;
```

- Callee runs

```
push fp;
fp := sp;
// allocate space for buffer
sp := sp + sizeof(buffer);
gets(buffer);
// user enters shellcode
// gets returns
sp := fp;
fp := pop();
pc := pop();
```



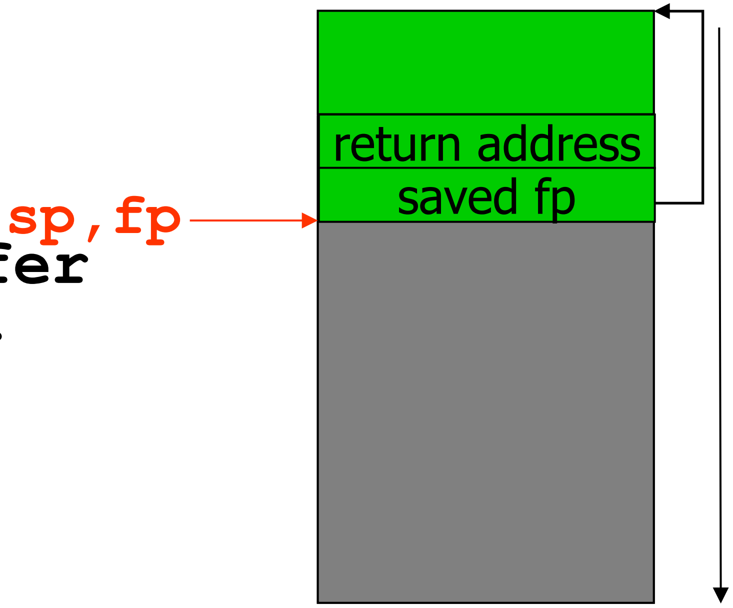
Stack smashing

- Caller runs

```
push return_address;
```

- Callee runs

```
push fp;
fp := sp;
// allocate space for buffer
sp := sp + sizeof(buffer);
gets(buffer);
// user enters shellcode
// gets returns
sp := fp;
fp := pop();
pc := pop();
```



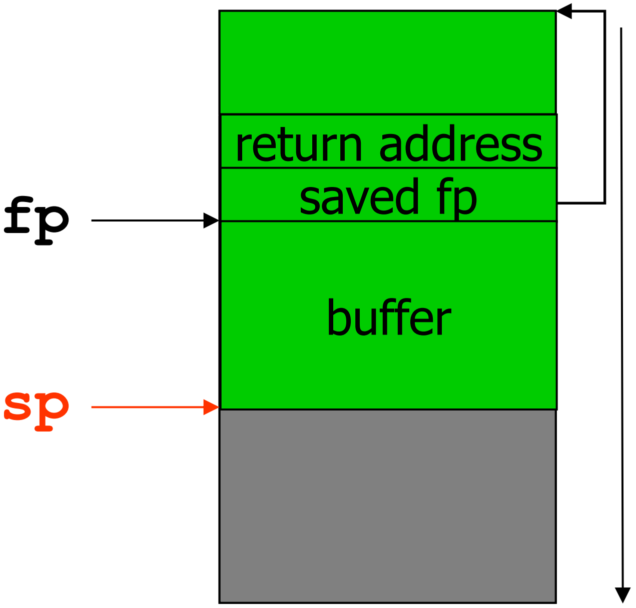
Stack smashing

- Caller runs

```
push return_address;
```

- Callee runs

```
push fp;
fp := sp;
// allocate space for buffer
sp := sp + sizeof(buffer);
gets(buffer);
// user enters shellcode
// gets returns
sp := fp;
fp := pop();
pc := pop();
```



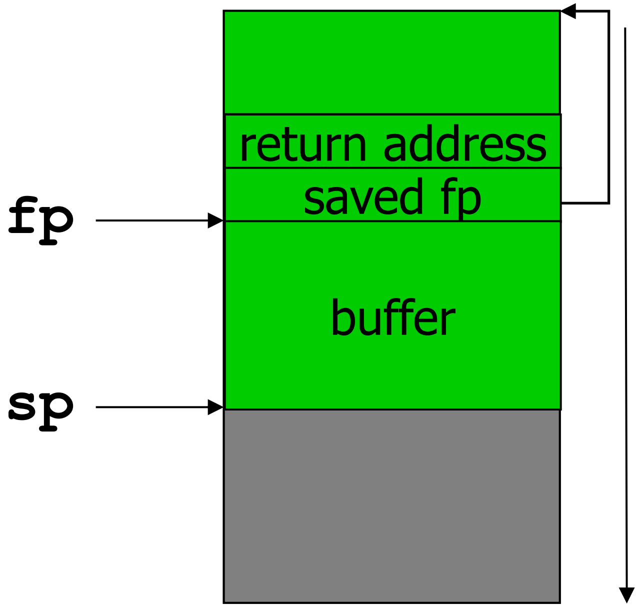
Stack smashing

- Caller runs

```
push return_address;
```

- Callee runs

```
push fp;
fp := sp;
// allocate space for buffer
sp := sp + sizeof(buffer);
gets(buffer);
// user enters shellcode
// gets returns
sp := fp;
fp := pop();
pc := pop();
```



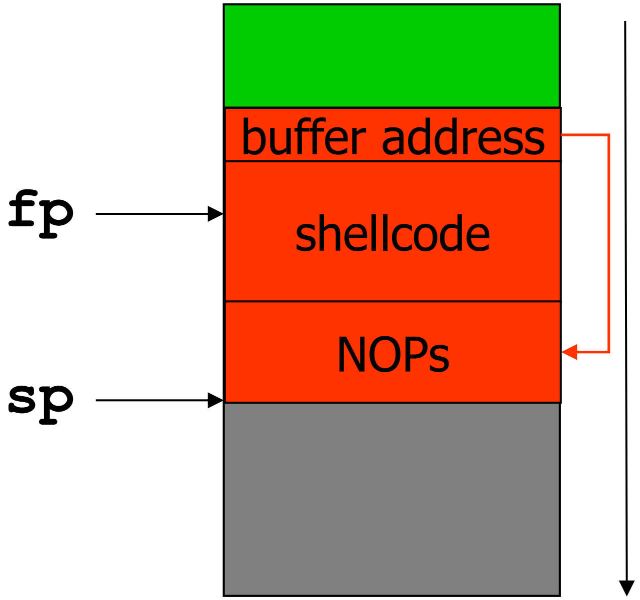
Stack smashing

- Caller runs

```
push return_address;
```

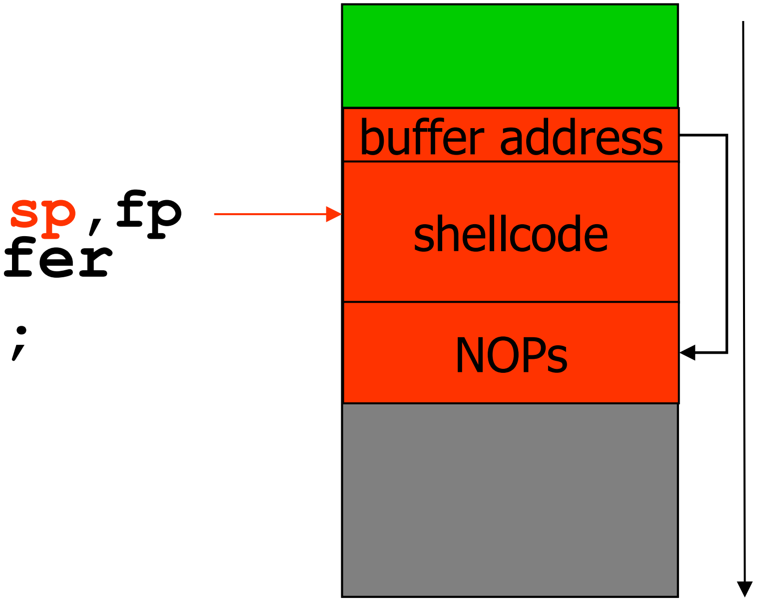
- Callee runs

```
push fp;
fp := sp;
// allocate space for buffer
sp := sp + sizeof(buffer);
gets(buffer);
// user enters shellcode
// gets returns
sp := fp;
fp := pop();
pc := pop();
```



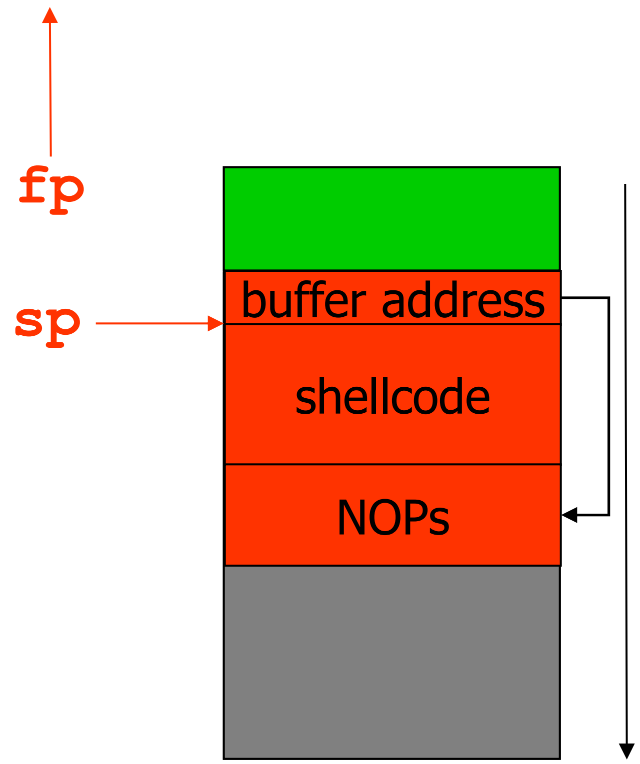
Stack smashing

- Caller runs
```
push return_address;
```
- Callee runs
```
push fp;
fp := sp;
// allocate space for buffer
sp := sp + sizeof(buffer);
gets(buffer);
// user enters shellcode
// gets returns
sp := fp;
fp := pop();
pc := pop();
```



Stack smashing

- Caller runs
  `push return_address;`
- Callee runs
  `push fp;`
  `fp := sp;`
  `// allocate space for buffer`
  `sp := sp + sizeof(buffer);`
  `gets(buffer);`
  `// user enters shellcode`
  `// gets returns`
  `sp := fp;`
  `fp := pop();`
  `pc := pop();`



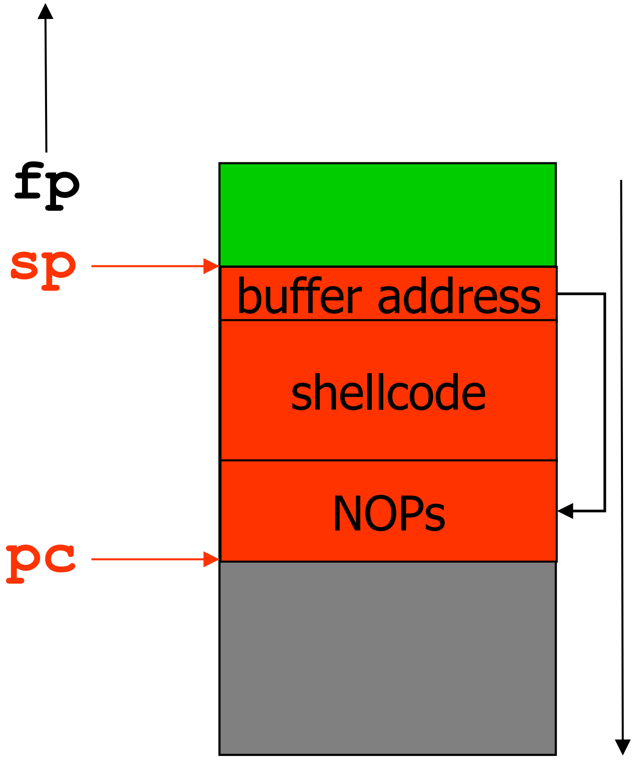
Stack smashing

- Caller runs

```
push return_address;
```

- Callee runs

```
push fp;
fp := sp;
// allocate space for buffer
sp := sp + sizeof(buffer);
gets(buffer);
// user enters shellcode
// gets returns
sp := fp;
fp := pop();
pc := pop();
```



Protection: coding practice

- Use type-safe languages when you can
- Type-safe dialects of C exist
  - Checked C by Microsoft
    - Need to provide annotations to arrays/pointers
    - Static/dynamic boundary and null pointer checks
  - Rust
    - developed by Mozilla Research
    - static typing
- If you have to use C
  - strcpy -> strncpy,…
  - Restrict the scope of elevated privileges
  - FORTIFY_SOURCE adds boundary checks

Protection: tools

- Static analysis
  - Clang static analyzer, Coverity, …
- Run-time protection with GCC/Clang
  - -fstack-protector flag adds canaries
  - Clang's SafeStack prevents all stack buffer overflows
- System support
  - Non-executable stack
  - Address obfuscation
  - Memory tagging in hardware

Database security

Database security

- Area by itself
- Basics
- Access control
- Views
- Field protection
- Statistical attacks
- SQL injection
  - see web application security lecture

Basics

- Encrypt connection between caller and db
- User authentication + access control (SQL)
- Protection
  - clearance 0-3

```
SELECT documents
  FROM docdb
WHERE clearance <=1
```

Access control in SQL

- `GRANT action(s) ON object TO user(s)`
- `REVOKE action(s) ON object TO user(s)`

- **Actions:** `SELECT, INSERT, DELETE, UPDATE`

Views

- "Virtual tables"

```
CREATE VIEW gen_employee_info AS
  SELECT eid, name, phone, email
   FROM employee_info
```

- Caution needed
  - Views implemented by temporary tables
    - can be slow
  - Clients should not connect directly to db
    - o/w a malicious caller may see entire db

Field protection

- Sensitive fields need to be protected besides access controls
  - credit card numbers
- Encryption does the job
  - decrypt on retrieval by authorized party
- How do you search encrypted data?
  - retrieve entire search space and search
- Secure information retrieval
    - direct search on encrypted data
    - search without revealing the query to db

Statistical attacks

- Need to protect individual records
  - salaries, grades,…
- Want to release aggregates
  - average, sum, max, min,…
- How do you prevent statistical attacks?

```
SELECT COUNT(*)
  FROM students
 WHERE city = “Åmål”
   AND course = “concurrent programming”
```

- Suppose result is 1

```
SELECT AVG(grade)
  FROM students
 WHERE city = “Åmål”
   AND course = “concurrent programming”
```

Solution attempt

- Refuse answering queries if they apply to fewer than 10 tuples

```
SELECT COUNT(*)
  FROM students
 WHERE course = “concurrent programming”
```

::160

```
SELECT COUNT(*)
  FROM students
 WHERE NOT (city = “Åmål”)
  AND course = “concurrent programming”
```

::159

- We see that the result is 1

```
SELECT AVG(grade)
  FROM students
 WHERE course = “concurrent programming”
```

::3.8

```
SELECT AVG(grade)
  FROM students
 WHERE NOT (city = “Åmål”)
   AND course = “concurrent programming”
```

::3.79

- (G1+…+G160)/160 = 3.8 (G1+…+G160-GÅmal)/159 = 3.79
- :: GÅmal=5

Defense against statistical attacks

- Analyze statistical inferences
- Restrict query language
  - only average grade for all participants
- Introduce fake tuples
- Give approximate answers
  - Åmål approximated by West Sweden
- Introduce noise
- Differential privacy

TAL: Typed Assembly Languages

TAL: basic type structure

$$type ::= \mathrm{int} \mid \mathrm{code}\{\, r_1{:}t_1,\ r_2{:}t_2,\ r_3{:}t_3,\ \ldots\}$$

A value with type code$\{$ **`r1`** $: t_1,$ **`r2`** $: t_2,$ **`r3`** $: t_3, \ldots\}$ must be a label, which when you jump to it, expects you to at least have values of the appropriate types in the corresponding registers.

Simple TAL program with types

```
fact:{r1:int,r2:int,r31:code{r1:int}}
   ; r1 = n, r2 = accum, r31 = return address
   sub r3, r1, 1
   ;{r1:int,r2:int,r31:code{r1:int},r3:int}
   ble r3, L2
   mul r2, r2, r1
   mov r1, r3
   jmp fact
L2:{r2:int, r31:code{r1:int}}
   mov r1, r2
   jmp r31
```

Badly typed program

```
fact:{r1:int,r31:code{r1:int}}
   ; r1 = n, r2 = accum, r31 = return address
   sub r3, r1, 1; {r1:int,r31:code{r1:int},r3:int}
   bge r1, L2
   mul r2, r2, r1        ; ERROR!  r2 doesn't have a
   type
   mov r1, r3
   jmp fact
L2:{r2:int, r31:code{r1:int}}
   mov r1, r2
   jmp r1   ; ERROR!  r1 is an integer!
```

Copyright protection and code obfuscation

Copyright protection

- Tough as any client-side security
- Trade-off with usability and performance
- Impact on privacy (“phone home” applications)
- Impossible to solve but there are tricks to raise the bar for the attacker

Copyright protection schemes

- License keys
  - Can be copied
  - Should not be reconstructable from program (could store key hash)
  - Key checks need to be tamperproof
- On-line Licenses
  - Requires on-line connection (firewall, privacy issues)
  - Checks should be tamperproof
  - Enterprise license servers (against casual pirates)

Tamperproofing

- Antidebugger measures
  - Make debugger crash (via cache)
- Checksums
- Responding to misuse
  - Add subtle bugs
  - Have bugs at start, automatically fix only if no misuse detected
  - Downside: endless support calls…
- Decoys
  - “naïve checks”
  - real checks elsewhere (for example checksums on naïve-check code)
  - Separate copies of license data

Code obfuscation

- Add code that never executes, or does nothing
  - Obvious calculations looking complex
- Move code around
  - Spread related functions
  - Copy and rename the same function instead of calling twice
- Encode your data oddly
  - Strange conversions
  - Encrypt data in memory and hide keys
- Downside
  - Horrible programming style
  - Efficiency
  - Usability/maintanability