IS893: Advanced Software Security

2. Basic Concepts

Kihong Heo

KAIST

Security Vulnerability

- A weakness that can be **exploited** by an attacker
  - design flaw, implementation bug, etc
- See CWE (Common Weakness Enumeration) and CVE (Common Vulnerabilities and Exposures)

CWE Definitions

Sort Results By : CWE Number Vulnerability Count

Total number of cwe definitions : 668 Page : 1 (This Page) 2 3 4 5 6 7 8 9 10 11 12 13 14

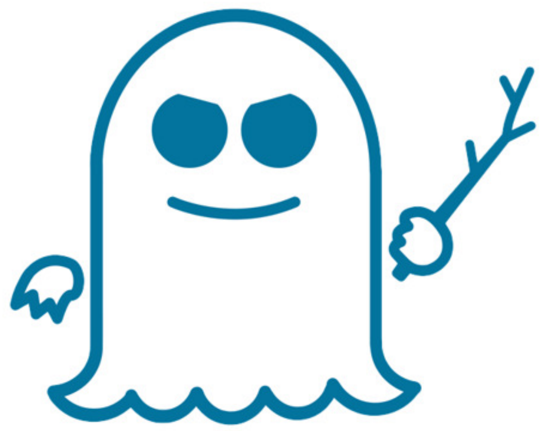
Select Select&Copy

CWE Number	Name	Number Of Related Vulnerabilities
119	Failure to Constrain Operations within the Bounds of a Memory Buffer	12328
79	Failure to Preserve Web Page Structure ('Cross-site Scripting')	11807
20	Improper Input Validation	7669
200	Information Exposure	6316
89	Improper Sanitization of Special Elements used in an SQL Command ('SQL Injection')	5643
22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	2968
94	Failure to Control Generation of Code ('Code Injection')	2400
125	Out-of-bounds Read	2122
287	Improper Authentication	1746
284	Access Control (Authorization) Issues	1627
416	Use After Free	1256
190	Integer Overflow or Wraparound	1113
476	NULL Pointer Dereference	900
78	Improper Sanitization of Special Elements used in an OS Command ('OS Command Injection')	788
787	Out-of-bounds Write	737
362	Race Condition	615
59	Improper Link Resolution Before File Access ('Link Following')	518
77	Improper Sanitization of Special Elements used in a Command ('Command Injection')	489
400	Uncontrolled Resource Consumption ('Resource Exhaustion')	463
611	Information Leak Through XML External Entity File Disclosure	393
434	Unrestricted Upload of File with Dangerous Type	385
732	Incorrect Permission Assignment for Critical Resource	350
74	Failure to Sanitize Data into a Different Plane ('Injection')	327
798	Use of Hard-coded Credentials	319
772	Missing Release of Resource after Effective Lifetime	306
269	Improper Privilege Management	305
601	URL Redirection to Untrusted Site ('Open Redirect')	265
502	Deserialization of Untrusted Data	257
134	Uncontrolled Format String	216
704	Incorrect Type Conversion or Cast	180
415	Double Free	173
522	Insufficiently Protected Credentials	158
532	Information Leak Through Log Files	133
326	Inadequate Encryption Strength	111
369	Divide By Zero	111
285	Improper Access Control (Authorization)	110
306	Missing Authentication for Critical Function	90
120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')	80
319	Cleartext Transmission of Sensitive Information	76
347	Improper Verification of Cryptographic Signature	76
770	Allocation of Resources Without Limits or Throttling	76
427	Uncontrolled Search Path Element	74
345	Insufficient Verification of Data Authenticity	68
617	Reachable Assertion	68
129	Improper Validation of Array Index	67
327	Use of a Broken or Risky Cryptographic Algorithm	65
668	Exposure of Resource to Wrong Sphere	63
276	Incorrect Default Permissions	60
404	Improper Resource Shutdown or Release	59
311	Missing Encryption of Sensitive Data	56

Heartbleed, 2014
OpenSSL
CVE-2014-0160

goto fail, 2014
MacOS / iOS
CVE-2014-1266

Shellshock, 2014
Bash
CVE-2014-6271

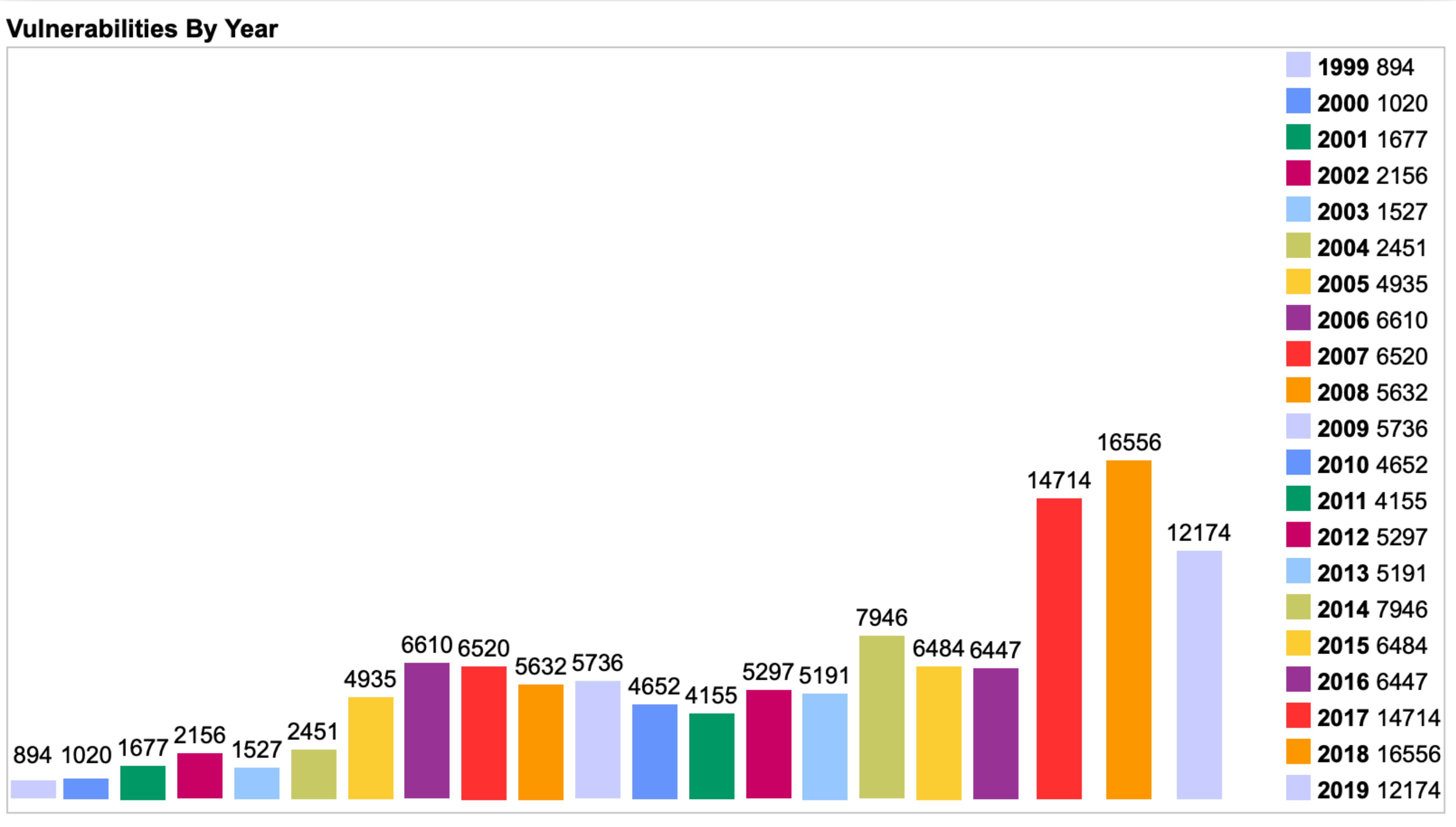


Spectre, 2017
Many CPUs
CVE-2017-5715
CVE-2017-5753

Meltdown, 2017
Many CPUs
CVE-2017-5754

CVEs Over Time

- Gradually increasing over time, why?
- More SW/HW, more bugs, and more powerful analysis tools!



Software Security

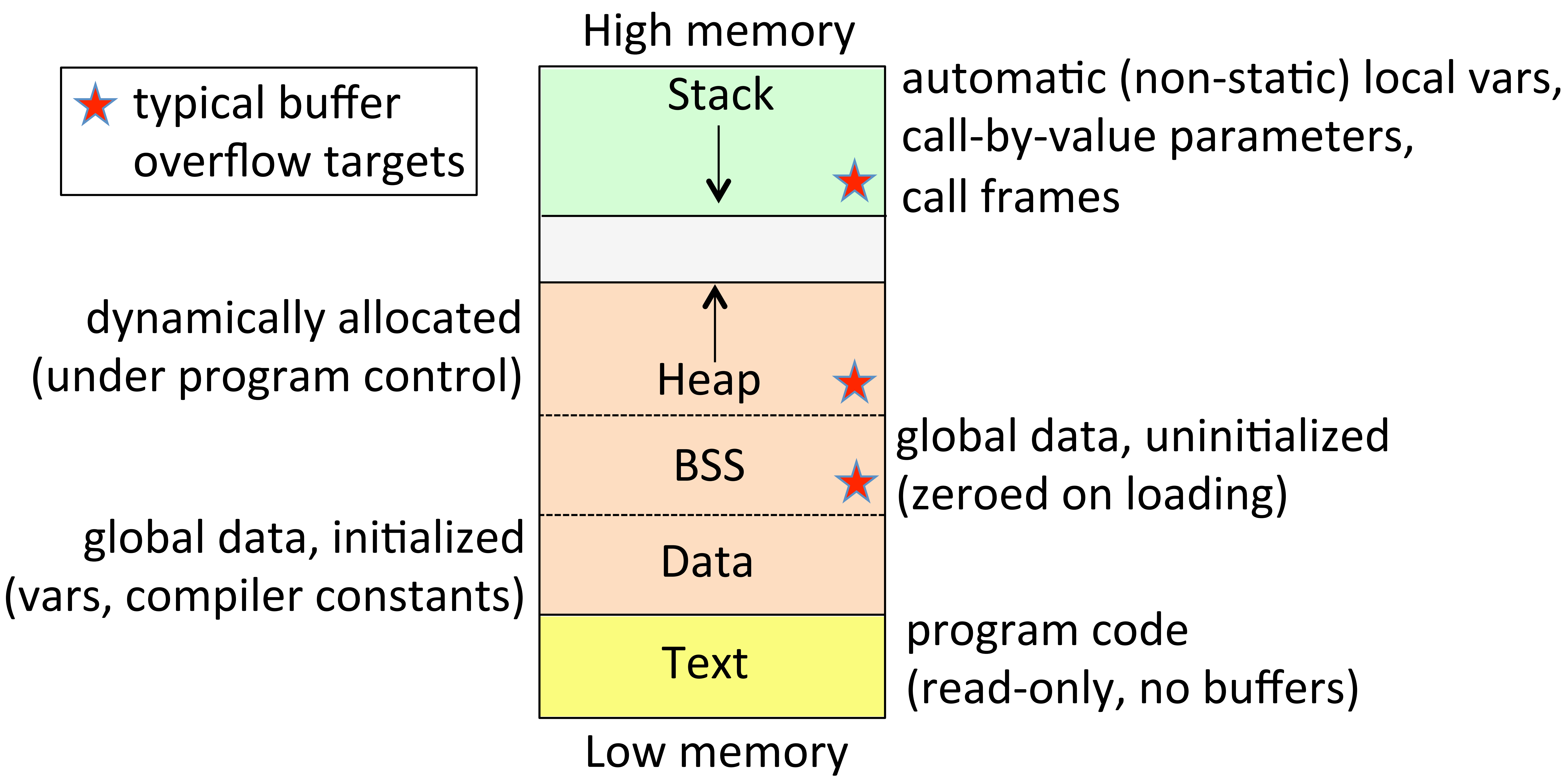
- Focus on **exploitable** software implementation errors and design flaws
- What happens if someone exploits security vulnerabilities?
  - privilege elevation, arbitrary code execution, access to all files, DoS, etc

Case Study: Buffer Overflows

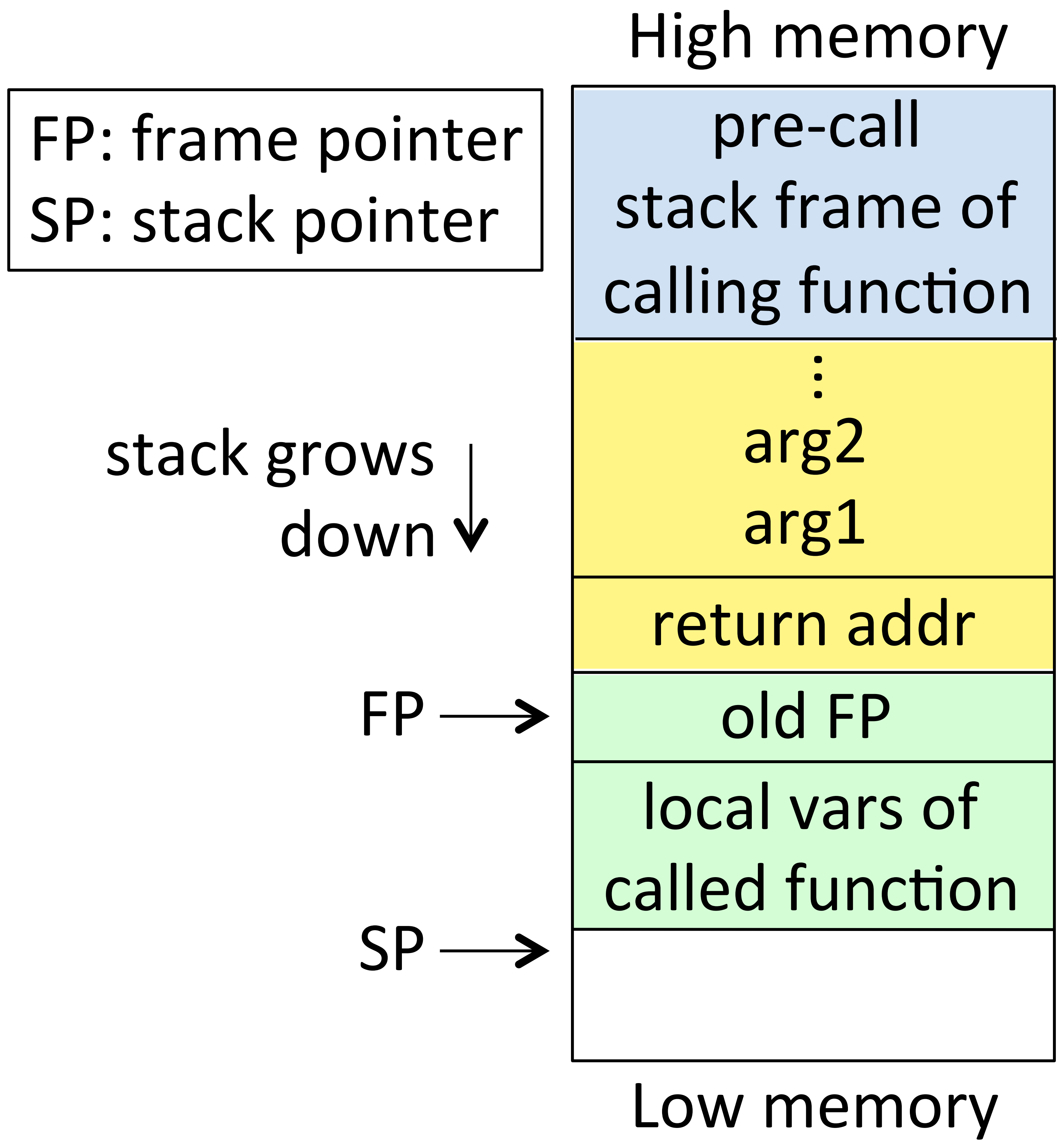
- Read or write more bytes to a buffer than allocated for it
- Common yet serious problems in languages like C/C++
- Unpredictable outcomes (i.e., undefined behavior)
    - E.g., crash, incorrect output, no effect, etc

```c
void myfunction(char *src) {
    int var1, var2;
    char var3[4];
    // what if the length of src > 3?
    strcpy(var3, src);
}
```

Common Memory Layout



User-space Stack and Calling Convention

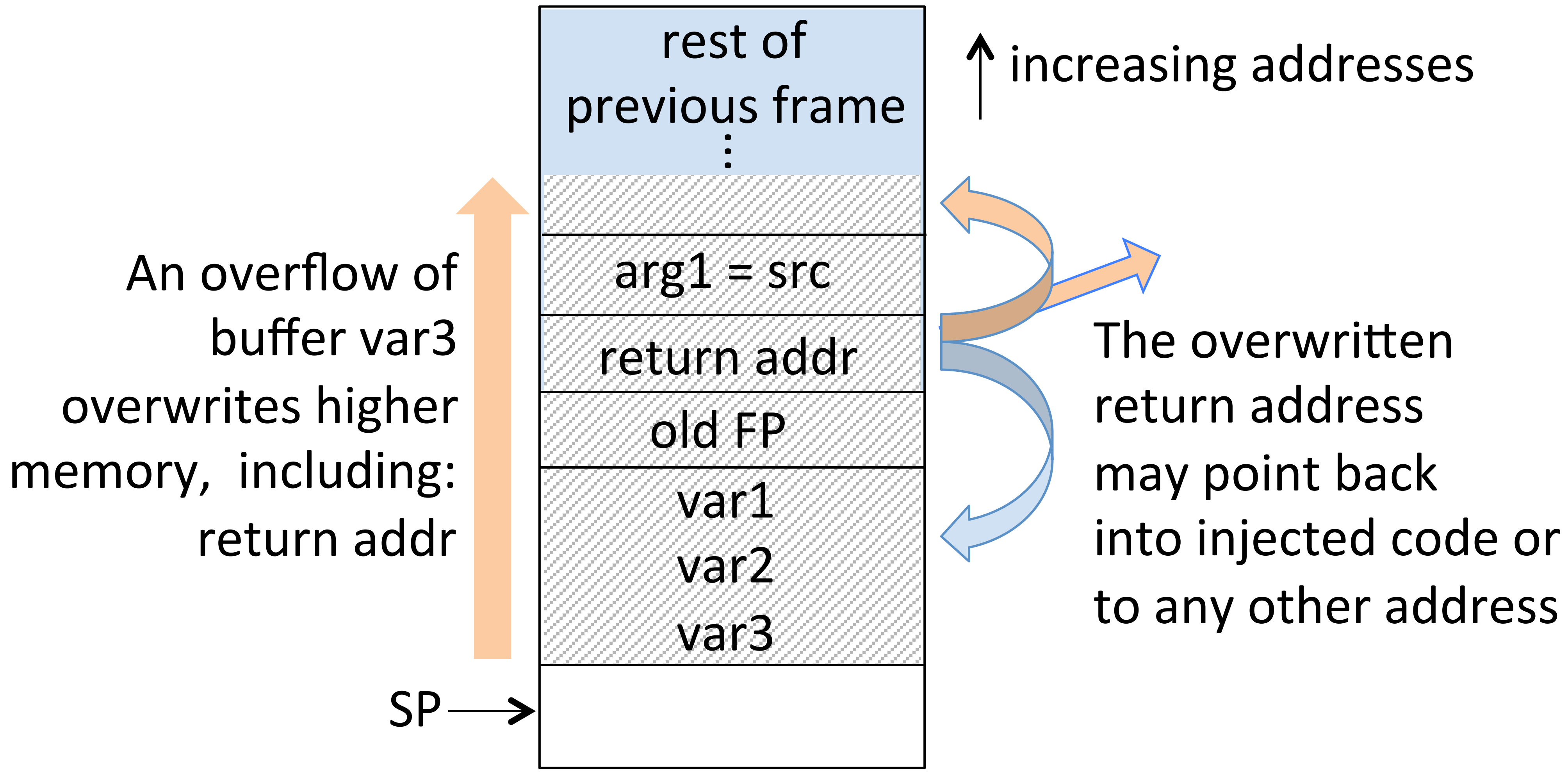


1. calling function pushes args onto stack
2. "call" opcode pushes Instruction Pointer (IP) as return address, then sets IP to begin executing code in called function
3. called function pushes FP for later recovery
4. FP ← SP (so FP points to old FP), now FP+k = args, FP-k = local vars
5. decrement SP, making stack space for local vars
6. called function executes until ready to return
7. called function cleans up stack before return (SP ← FP , FP ← old FP popped from stack)
8. "ret" opcode pops return address into IP, to resume execution back to calling function

Stack-based Buffer Overflow

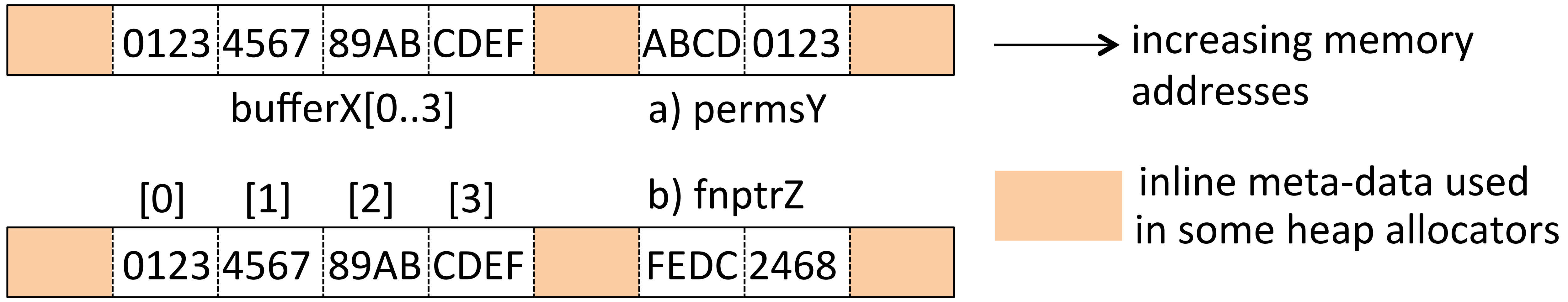
- Buffer overflows on stack can overwrite higher memory
  - Esp., return address
- Why is return address important?
  - Control-flow hijacking!

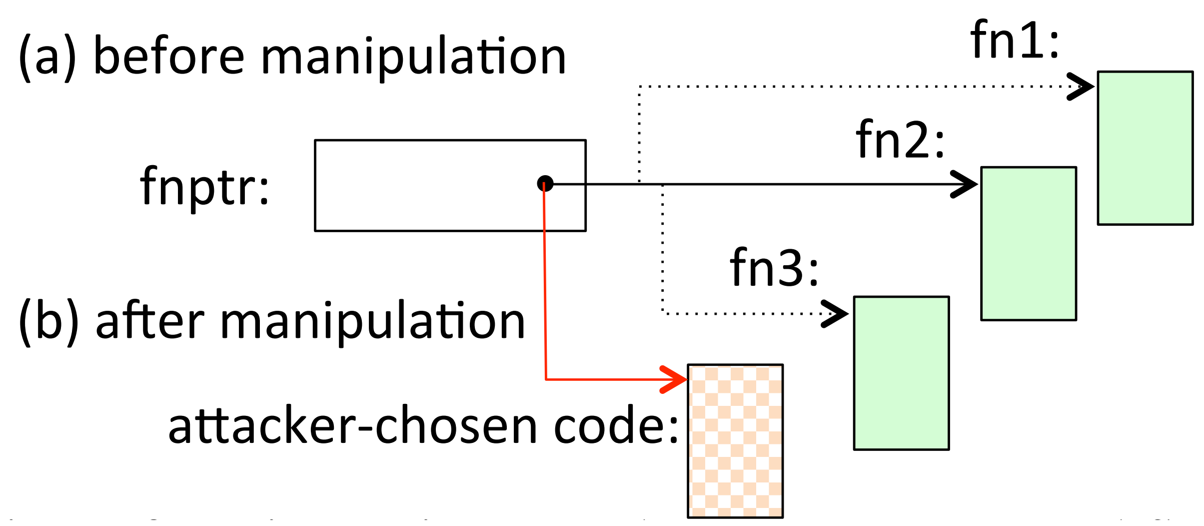
```
void myfunction(char *src) {
    int var1, var2;
    char var3[4];
    // what if the length of src > 3?
    strcpy(var3, src);
}
```



Heap-based Buffer Overflow

- Find an exploitable buffer and a strategically useful variable for an attack
- Corrupt important data such as access control or function pointers





Effect of Buffer Overflows

- Program control-flow can be directly altered by corrupting data
  - stack-based pointers (e.g., return addresses, frame pointers)
  - function pointers, jump table, etc
  - addresses used in setjmp/longjmp
  - (indirectly) by curating data used in a branching test

Generic Exploit Steps

1. **Inject or locate code** that the attacker desires to be executed within the target program's address space

2. **Corrupt** control flow data (e.g., by a buffer overflow)

3. **Transfer** program control flow to the target code of step 1

Case Study: Integer-based Vulnerabilities

- Exploitable code sequences due to integer bugs
  - E.g., unsafe type casting, integer overflow, etc

```
BOOL handle_login(userid, password) {
  attempts = attempt + 1;
  // what if “attempts” overflows?
  if (attempts <= MAX_ALLOWED) {
    if (pswd_is_ok(userid, password) {
      attempts = 0;
      return TRUE;
    }
  }
  return FALSE;
}
```

```
void init_table() {
  unsigned int width = input();
  unsigned int height = input();
  // what if “width * height” overflows?
  table = malloc(width * height);
  … table[i][j] …
}
```

Consequences of Integer Vulnerabilities

- Unexpected subscript values enable access to unintended addresses
- Smaller than anticipated integer values result in under-allocation of memory
- Underflow → neg size-arg to malloc → large pos integer → out-of-memory
- Overflow → large neg integer → compared to an upper bound of a loop → excessive number of iterations
- Etc

Defenses

- How to protect your software and systems?
- Can we estimate all potential vulnerabilities in advance?
- No general and perfect solution!
- Why?

A Hard Limit: Undecidability

Theorem (Rice's theorem). Any non-trivial semantic properties are undecidable.

- Non-trivial property: worth the effort of designing a program analyzer for
  - trivial: true or false for all programs
- Undecidable? If decidable, it can solves the Halting problem
  - An analyzer **A** for a property: "This program always prints 1 and finishes"
  - Given a program **P**, generate "**P**; print 1;"
  - **A** says "Yes": **P** halts, **A** says "No": **P** does not halt

In Practice

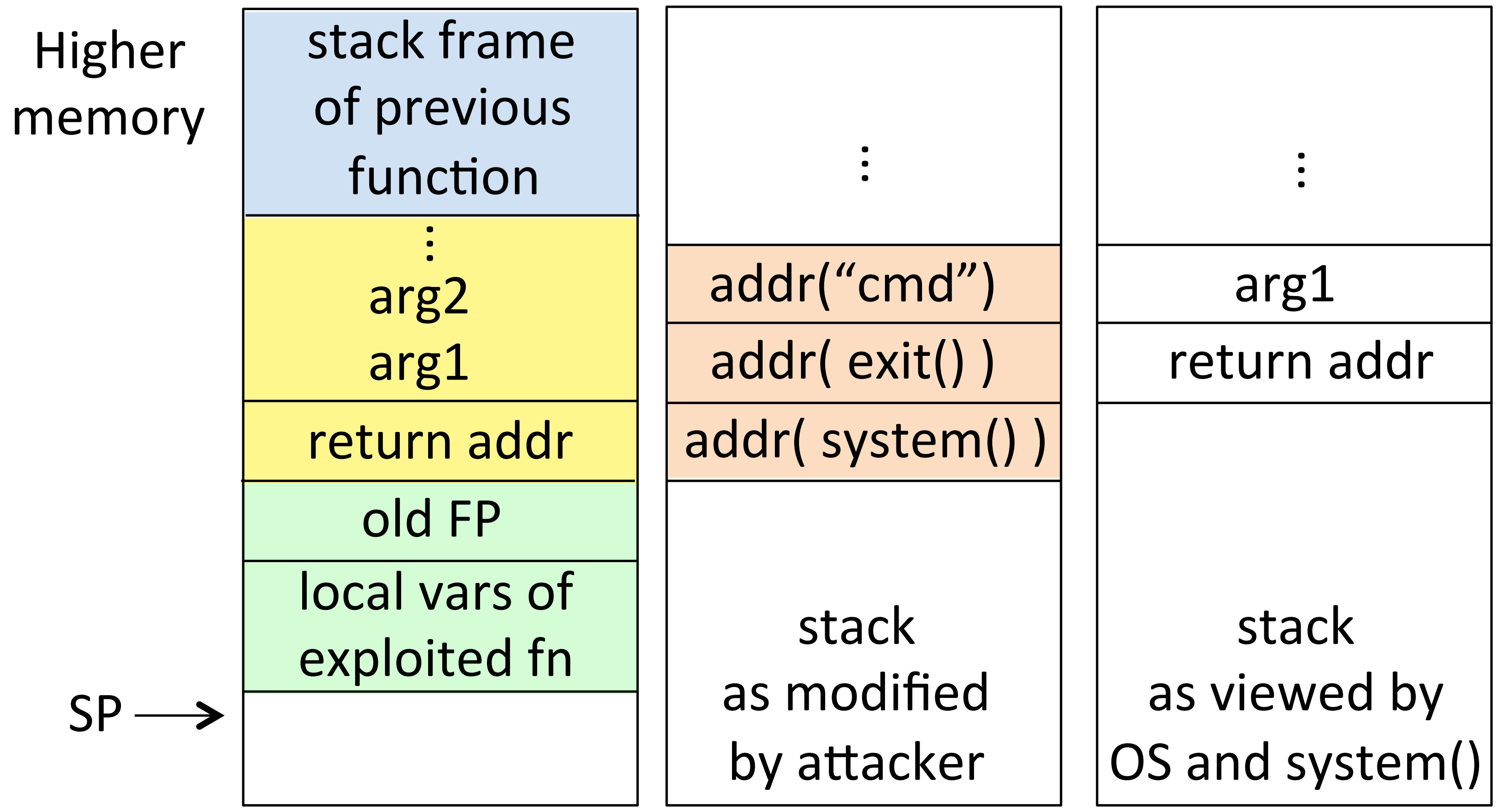
- Many approaches that alleviate the hard limit
  - Pre- / post-deployment approaches
  - Static / dynamic approaches
- Collaboration of multiple systems
  - Hardware, OS, compiler, program analysis, etc

Non-executable Memory

- Certain address ranges are marked invalid for execution by OS or hardware
  - E.g., stack, heap, BSS, etc
- However, does not always guarantee its use
  - E.g., backwards compatibility, disabled by an attacker, use of JIT
- Can prevent execution but not overwrite: indirect attacks are possible
  - E.g., return-to-libc

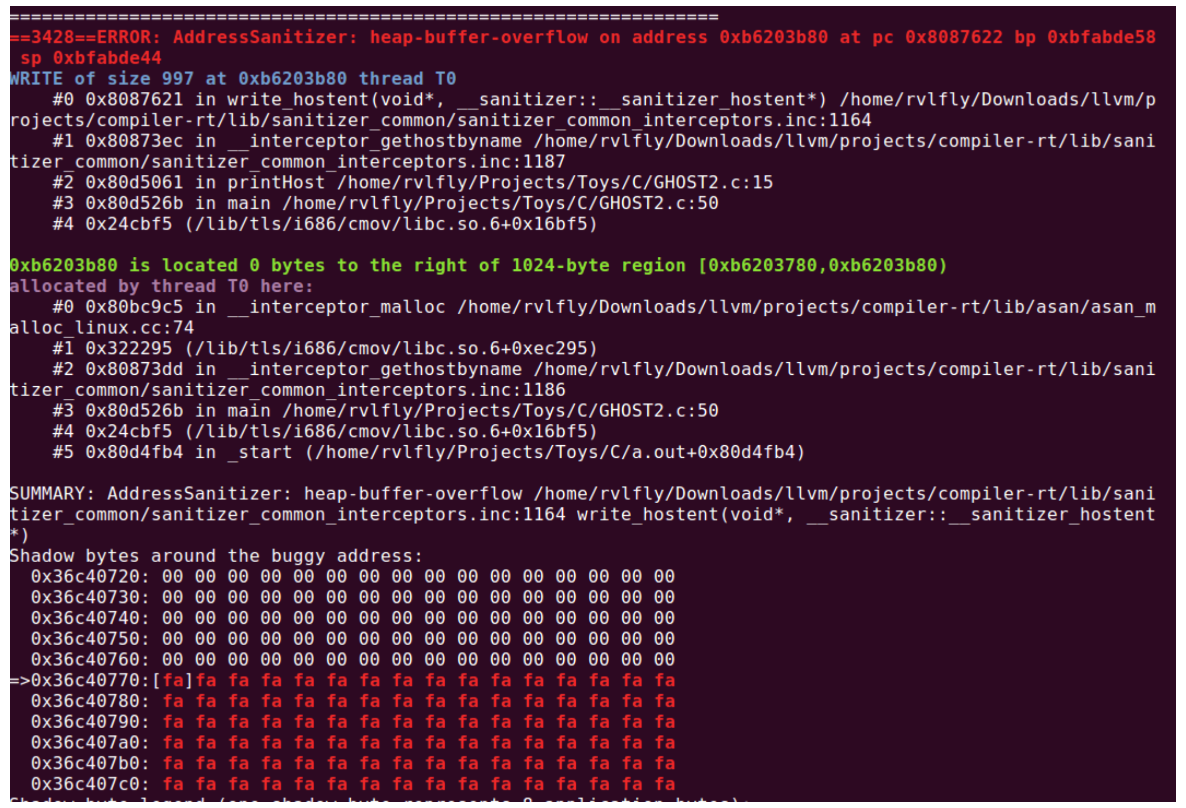
C.f., Return-to-libc

- Instead of injecting code into stack or heap, pass to existing system code
  - E.g., system calls or standard library functions in libc
- For example, pass code to `system()`



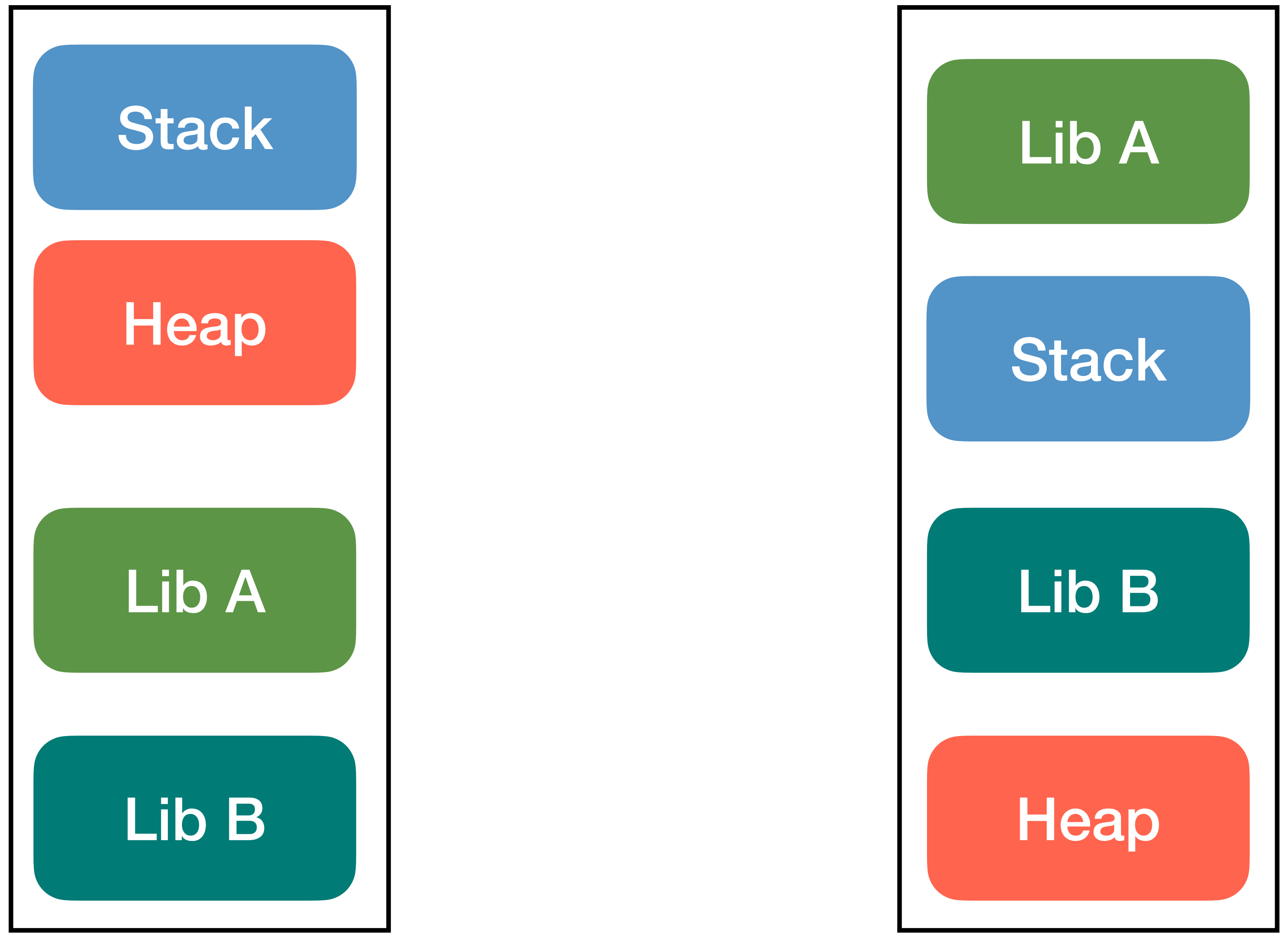
Runtime Checking

- Compilers instrument code to invoke checking code
  - E.g., add bound-checking code before every buffer access
- See LLVM's sanitizers (ASAN, UBSAN, MEMSAN, etc)
- Involve compiler support, runtime supports, and runtime overhead



Memory Layout Randomization

- Randomize the base address of the stack, heap, code, etc
  - E.g., randomly assign the base address of the segment of `libc`
- Introduced by Linux's PaX project, now in many mainstream OSs



Safe Languages & Libraries

- Unsafe languages: unsafe type casting, unchecked pointer arithmetic, etc
  - Use safe languages like OCaml, Rust, etc rather than C/C++
- Unsafe libraries: unsafe bound checking, etc
  - Use `strncpy` rather than `strcpy`
- Hurdles: legacy code

x

"*Over the course of its lifetime, there have been* ***69 security bugs*** *in* ***Firefox****'s style component. If we'd had a time machine and could have* ***written this component in Rust*** *from the start,* ***51 (73.9%) of these bugs*** *would not have been possible.*"
- Diane Hosfelt (SW Engineer at Mozilla), 2019

x

"*As our system has grown, a lot of the logic in our Ruby system sort of replicates a* ***type system****. … It is a shame to have to write all that when there is a solution that has existed in the world of programming languages for decades now.*"
- Alex Payne (API Lead at Twitter), 2009

x ERLANG

"***WhatsApp*** *uses a surprisingly* ***small amount of engineers*** *for the billions of users it caters to on a daily basis. How do they manage this?* ***Erlang*** *was built to solve very specific problems, in particular scaling a large system with it still remaining* ***highly reliable****.*"
- Erlang Solutions, 2018

x OCaml

"*There is, however, a* ***surprisingly wide swath of bugs*** *against which the* ***type system*** *is effective, including many bugs that are quite hard to get at through testing.*"
- Yaron Minsky (SW Engineer at Janestreet), 2018

Program Debloating

- More code = more vulnerable!
- Aggressively remove unnecessary code so that reduce attack surface
- For example, GNU tar with 97 cmd line options

Global variable declarations removed

Code containing CVE removed

Overwriting functionalities removed

```
int absolute_names;
int ignore_zeros_option;
struct tar_stat_info stat_info;

char *safer_name_suffix (char *file_name, int link_target) {
    int prefix_len;
    char *p;

    if (absolute_names) {
        p = file_name;
    } else {
      /* CVE-2016-6321 */
      /* Incorrect sanitization if "file_name" contains ".." */
      ...
    }
    ...
    return p;
}

void extract_archive() {
    char *file_name = safer_name_suffix(stat_info.file_name, 0);
    /* Overwrite "file_name" if exists */
    ...

void list_archive() { ... }
```

```
void read_and(void *(do_something)(void)) {
    enum read_header status;
    while (...) {
        status = read_header();
        switch (status) {
        case HEADER_SUCCESS: (*do_something)(); continue;
        case HEADER_ZERO_BLOCK:
          if (ignore_zeros_option) continue;
          else break;
        ...
        default: break;
        }
    }
    ...
}

/* Supports all options: -x, -t, -P, -i, ... */
int main(int argc, char **argv) {
    int optchar;
    while (optchar = getopt_long(argc, argv) !=
        switch(optchar) {
        case 'x': read_and(&extract_archive); break;
        case 't': read_and(&list_archive); break;
        case 'P': absolute_names = 1; break;
        case 'i': ignore_zeros_option = 1; break;
        ...
        }
    }
    ...
}
```

Unnecessary functionalities removed

Unsupported options removed

Program Analysis

- Over- or under-**approximate** program behavior
- Static approaches: without running programs
  - Approximately compute all possible program states
  - May have spurious results
- Dynamic approaches: with running programs
  - Concretely enumerate and observe program states
  - May not cover all possible behavior

Conclusion

- Software security can affect physical & data security
  - SW can manipulate machines and read / write data
- Growing interest as SW is eating the world!
  - Traditional SW: financial, military, privacy, etc
  - Emerging concerns: security of AI such as fairness or morality