Fundamentals of Computer Security

Fall 2022

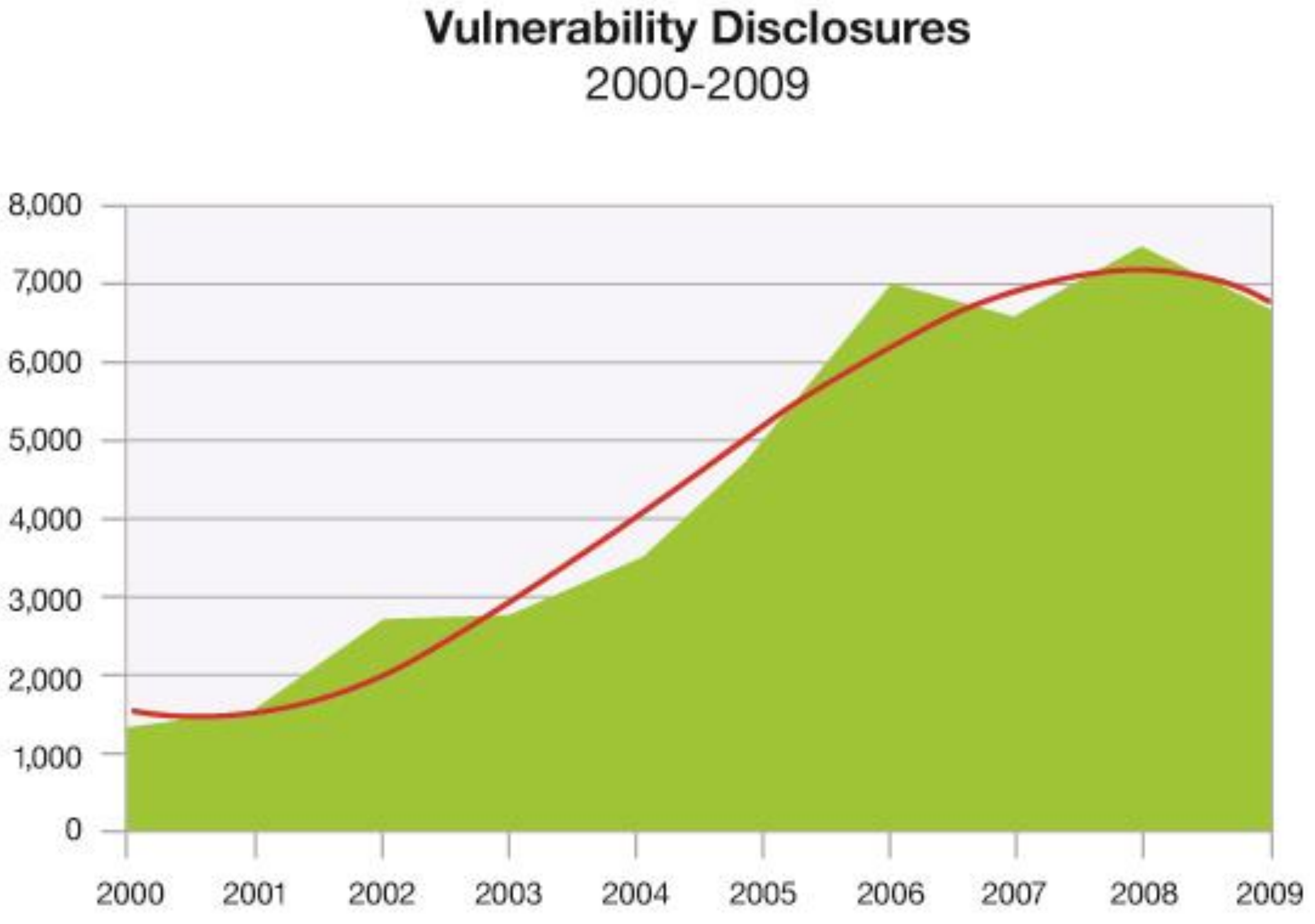
Radu Sion

Software Errors
Buffer Overflow
TOCTTOU

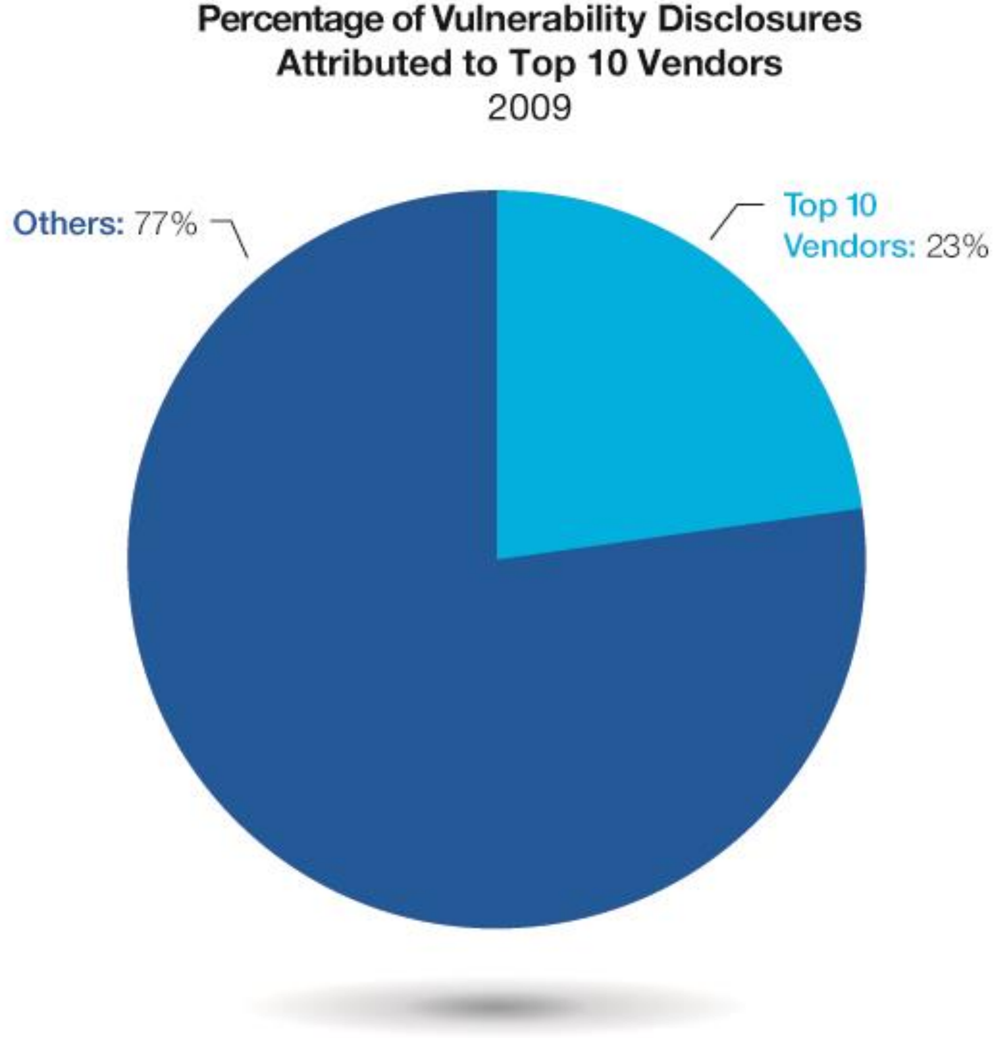
Why Security Vulnerabilities?

- Some contributing factors
  - Few courses in computer security ☺
  - Programming text books do not emphasize security
  - Few security audits
  - *C is an unsafe language*
  - Programmers have many other things to worry about
  - Consumers do not care about security
  - Security is expensive and takes time

Trends

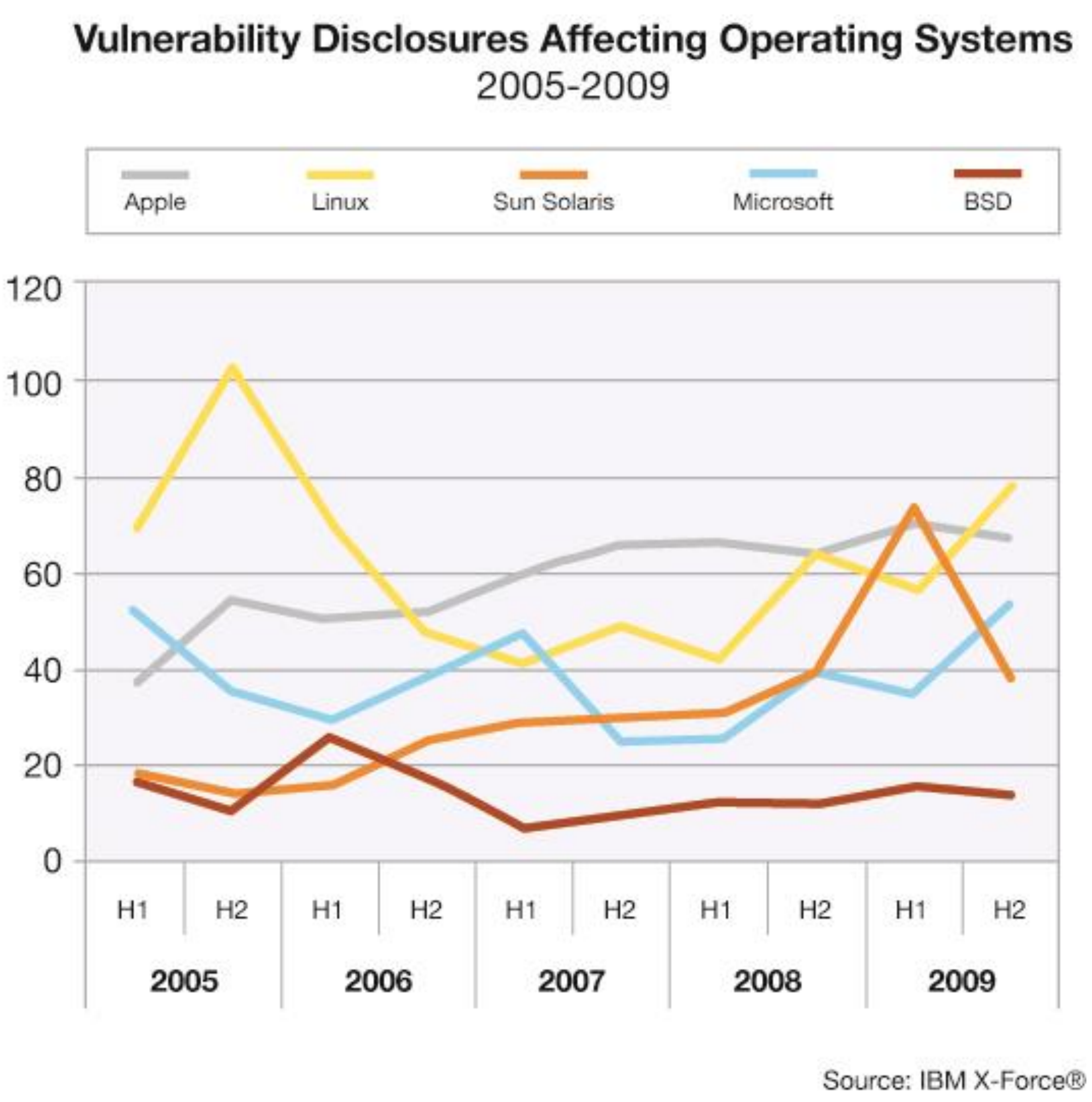


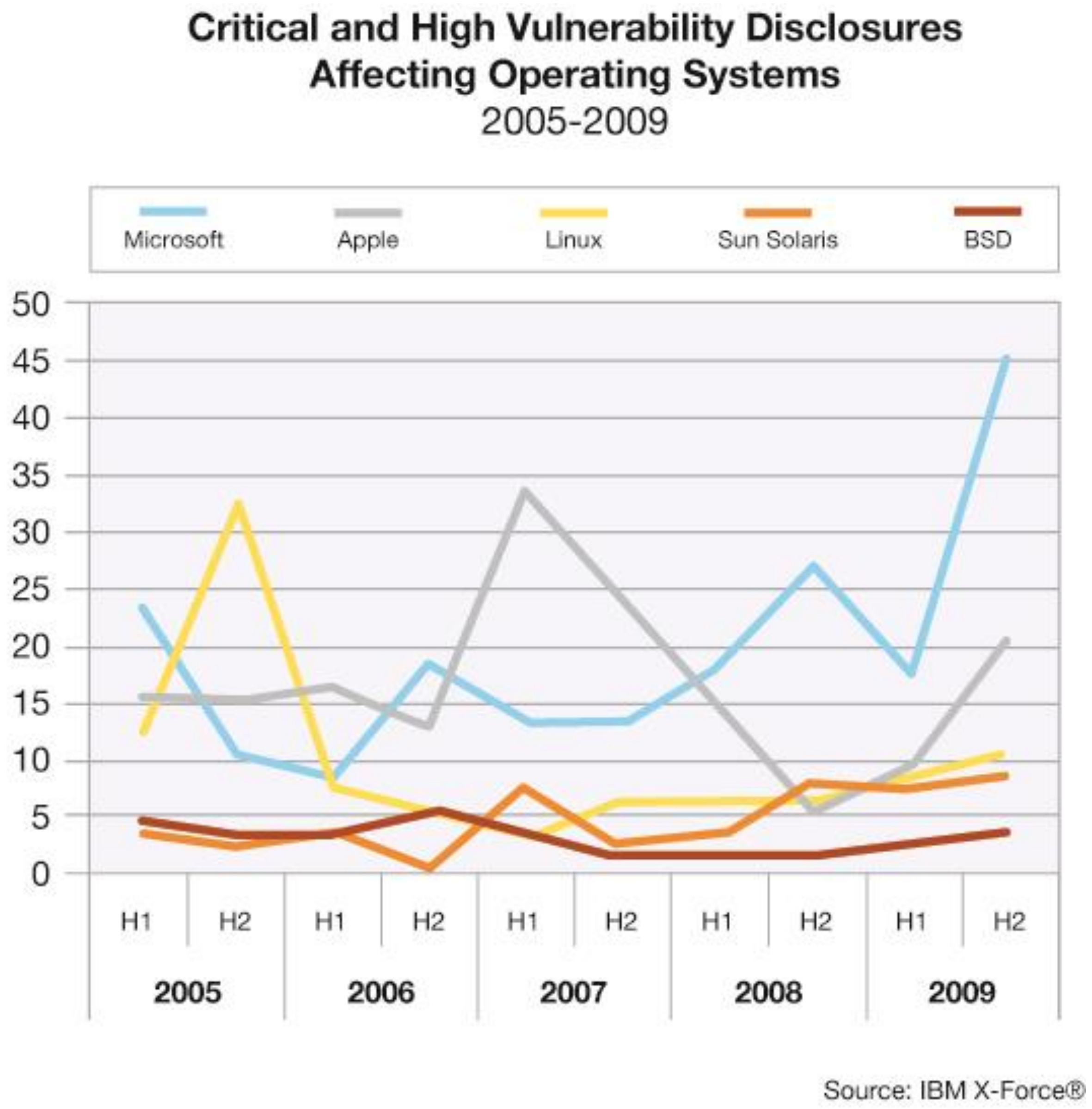
Source: IBM X-Force®



Source: IBM X-Force®

OS Vulnerabilities





Non-malicious Errors

- How to determine *quality* of program ?
  - Testing …
  - Number of faults in requirements, design and code inspections
- Example
  - Module A had 100 faults discovered and fixed
  - Module B had only 20
  - Which one is better ?
  - Software testing result: software with more faults is likely to have even more !!!

Fixing Faults

- Penetrate and Patch
  - Special teams test programs and find faults
  - If no attack found, the program was OK
  - Otherwise, not – More frequently
  - Then fix faults
- Problem: *The system became less secure !*
  - Focus on fixing the fault and not its context
  - Fault had side effects in other places
  - Fixing fault generated faults somewhere else
  - Fixing fault would affect functionality or performance

How many bugs/line of code

Up to 5% BPLOC!!!

Buffer Overflow Hall of Fame

- Morris worm (1988): overflow in fingerd
  - *6,000 machines infected (10% of existing Internet)*
- CodeRed (2001): overflow in MS-IIS web server
  - **Internet Information Services (IIS)**
  - Web server application
  - The most used web server after Apache HTTP Server
  - *300,000 machines infected in 14 hours*
- SQL Slammer(2003): overflow in MS-SQL server
  - *75,000 machines infected in **10 minutes** (!!)*

Buffer Overflow Hall of Fame (2)

- Sasser (2004): overflow in Windows LSASS
  - **Local Security Authority Subsystem Service**
    - Process in Windows OS
    - Responsible for enforcing the security policy on the system.
    - Verifies users logging on to a Windows computer or server, handles password changes, and creates access tokens
  - *Around 500,000 machines infected*

- Conficker (2008-09): overflow in Windows Server
  - *~10 million machines infected*

Memory Exploits

- Buffer is a data storage area inside computer memory (stack or heap)
  - Intended to hold pre-defined amount of data
- If executable code is supplied as “data”, victim’s machine may be fooled into executing it
- Code will give attacker control over machine

e.g. stack buffer

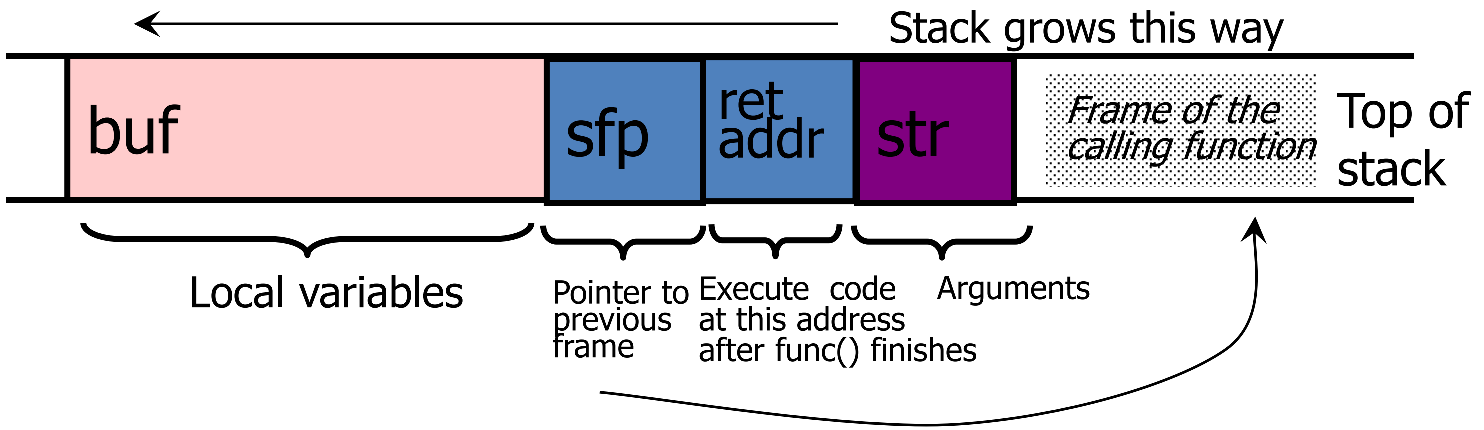
- Suppose Web server contains this function

```
void func(char *str) {
      char buf[126];
      strcpy(buf,str);
}
```

Allocate local buffer (126 bytes reserved on stack)

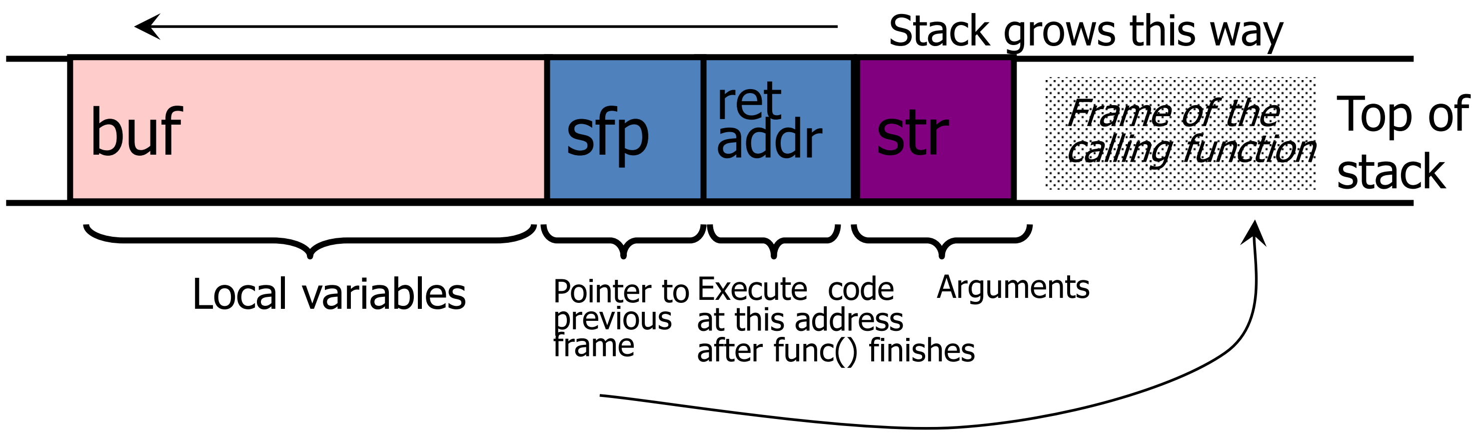
Copy argument into local buffer

- When this function is invoked, a new frame with local variables is pushed onto the stack



Stack buffer (2)

- When func returns
  - The local variables are popped from the stack
  - The old value of the stack frame pointer (sfp) is recovered
  - The return address is retrieved
  - The stack frame is popped
  - Execution continues from return address (calling function)



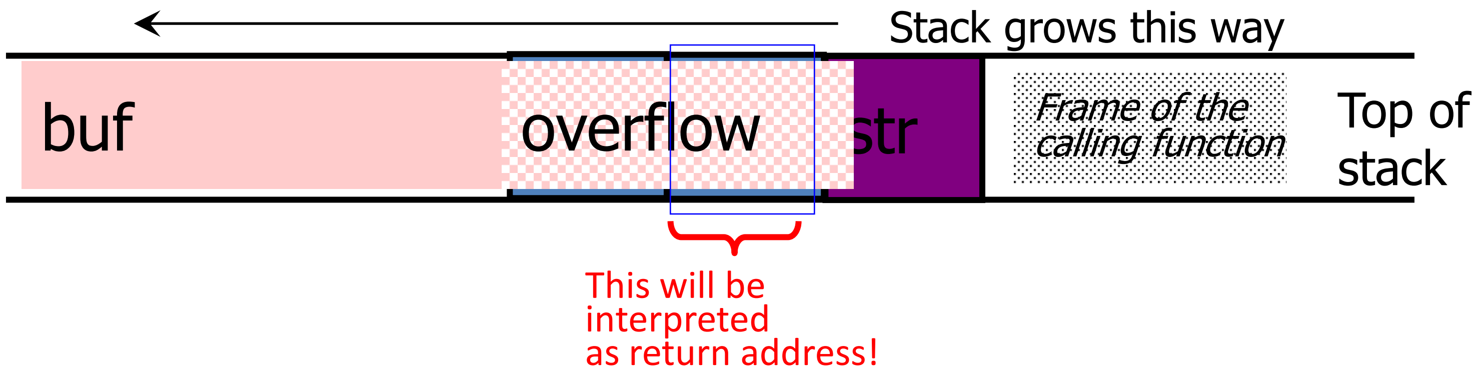
What if Buffer is Over-stuffed? ☺

- Memory pointed to by str is copied onto stack…

```
void func(char *str)
       char buf[126];
       strcpy(buf,str);
}
```

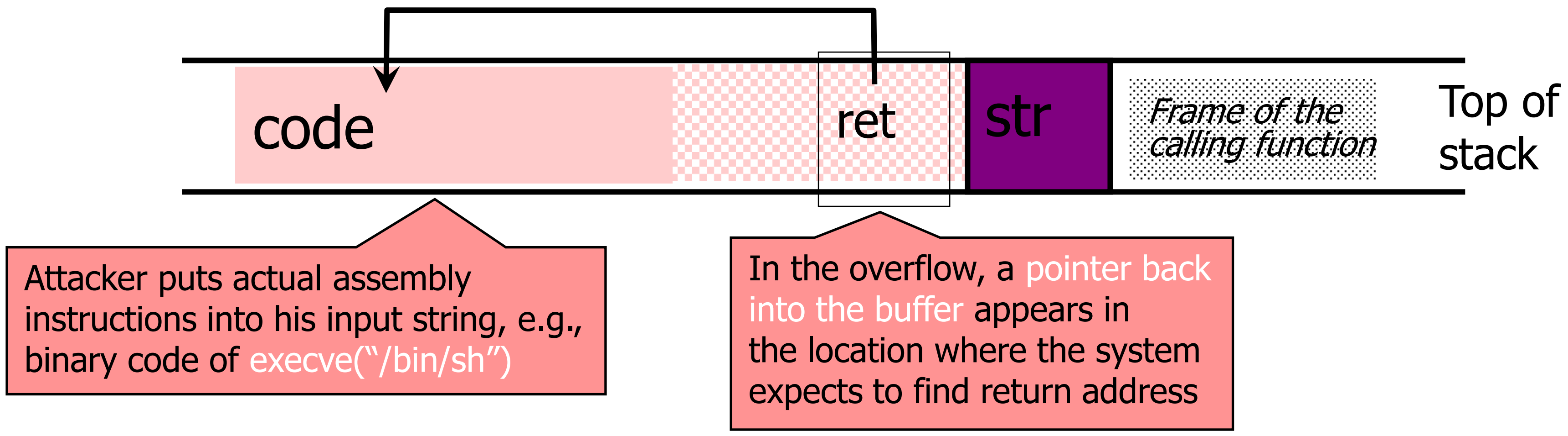
strcpy does NOT check whether the string at *str contains fewer than 126 characters

- If a string longer than 126 bytes is copied into buffer, it will overwrite adjacent stack locations



Attack 1: Stack Smashing

- Suppose buffer contains attacker-created string
  - For example, *str contains a string received from the network as input to some network service daemon



When function exits, code in the buffer will be executed, giving attacker a shell

Root shell if the victim program is setuid root

Buffer Overflow Difficulties

- Executable attack code is stored on stack, inside the buffer containing attacker's string
  - Stack memory is supposed to contain only data, but…
- For the basic attack, overflow portion of the buffer must contain *correct address of attack code* in the RET position
  - The value in the RET position must point to the beginning of attack assembly code in the buffer
  - Otherwise application will give segmentation violation
  - Attacker must correctly guess in which stack position his buffer will be when the function is called

Real Problem: No Range Checks

- strcpy does not check input size
  - strcpy(buf, str) simply copies memory contents into buf starting from *str until “\0” is encountered, ignoring the size of area allocated to buf
- Many C library functions are unsafe
  - strcpy(char *dest, const char *src)
  - strcat(char *dest, const char *src)
  - gets(char *s)
  - scanf(const char *format, …)
  - printf(const char *format, …)

Does range checking help?

- strncpy(char *dest, const char *src, size_t n)
  - If strncpy is used instead of strcpy, no more than n characters will be copied from *src to *dest
  - Programmer has to supply the right value of n
- Potential overflow in htpasswd.c (Apache 1.3):

```
… strcpy(record, user);
    strcat(record, ”:”);
strcat(record, cpw); …
```

Copies username ("user") into buffer ("record"), then appends ":" and hashed password ("cpw")

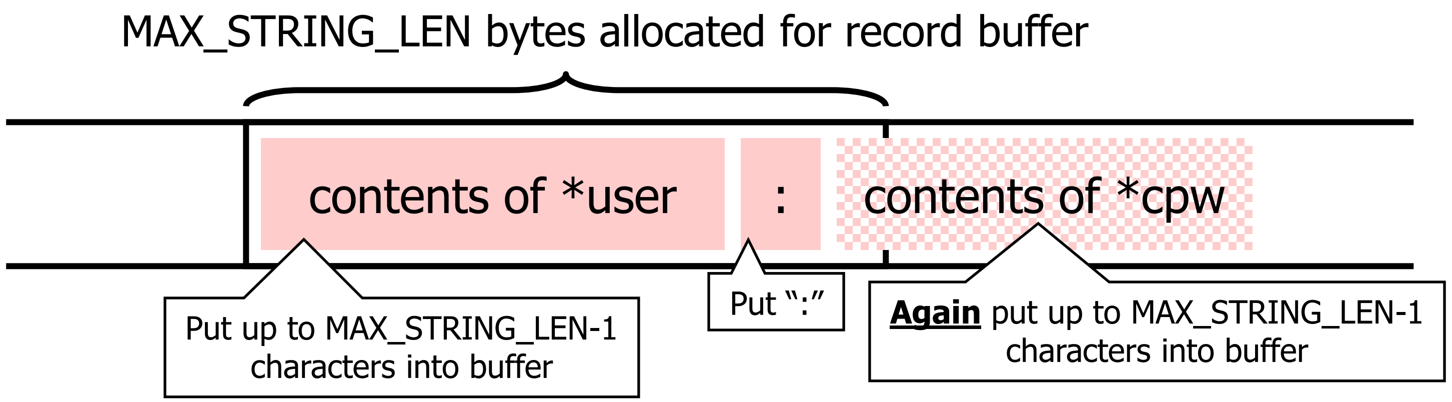
- Published "fix" (do you see the problem?):

```
… strncpy(record,user, MAX_STRING_LEN-1);
strcat(record,”:”);
strncat(record,cpw, MAX_STRING_LEN-1); …
```

“Fix”?

Published “fix” for Apache htpasswd overflow:

```
… strncpy(record,user, MAX_STRING_LEN-1);
    strcat(record, ”:”);
    strncat(record,cpw, MAX_STRING_LEN-1); …
```

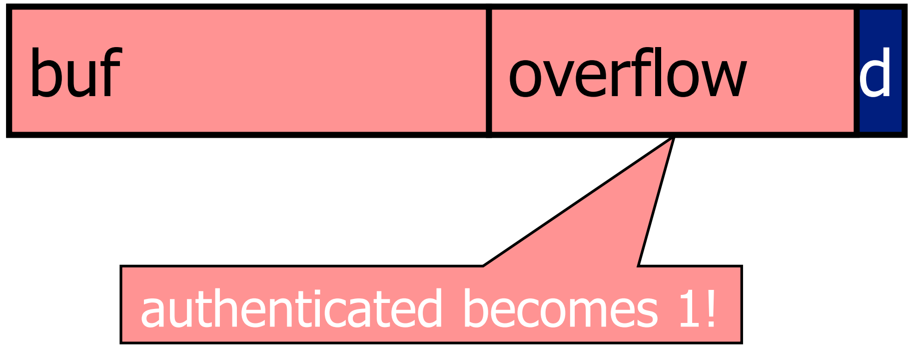


Attack 2: Variable Overflow

Somewhere in the code authenticated is set only if login procedure is successful

- Other parts of the code test authenticated to provide special access

```
char buf[80];
int authenticated = 0;
void vulnerable() {
  gets(buf);
}
```



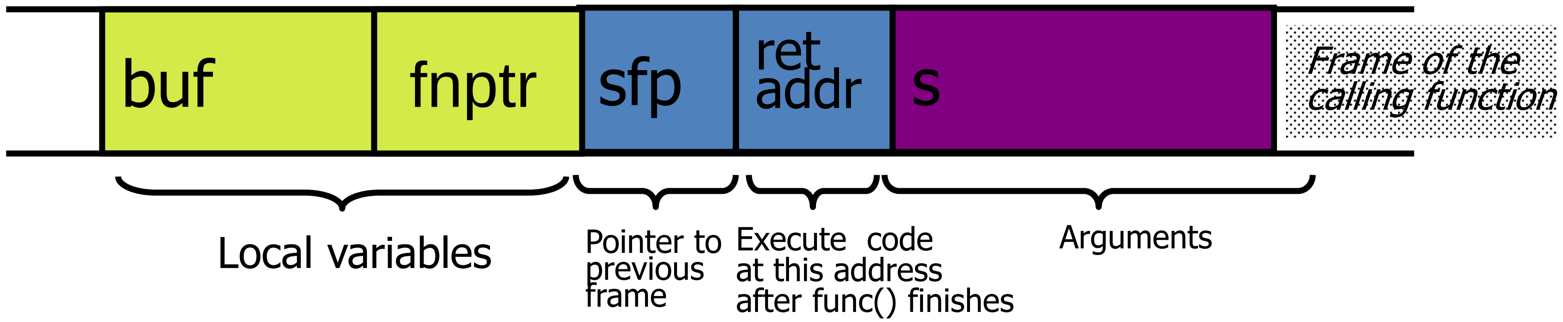
Attacker passes 81 bytes as input to buf

Attack 3: Alter Pointer Variables

fnptr is invoked somewhere else in the program

This is only the definition

```
void func(char *s){
  char buf[80];
  int (*fnptr)();
  gets(buf);
}
```



Alter Pointer Variables (2)

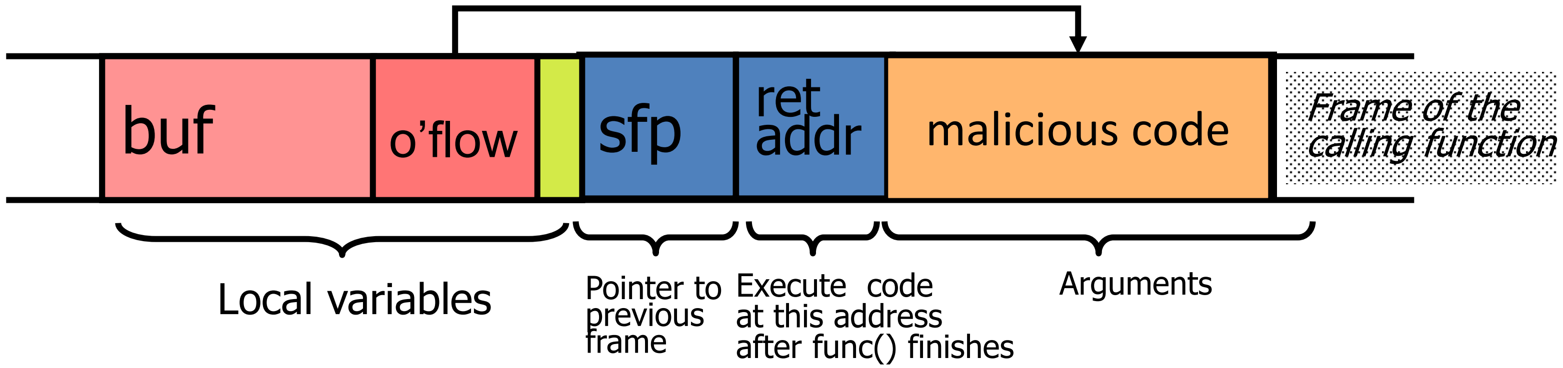

```
void func(char *s){
  char buf[80];
  int (*fnptr)();
  gets(buf);
}
```

Send malicious code in s

Overflow fnptr

- Pass more than 80 bytes in gets
- fnptr now points to malicious code

When fnptr is executed, malicious code is executed !



Attack 4: Frame Pointer


```
void func(char *s){
  char buf[80];
  gets(buf);
}
```

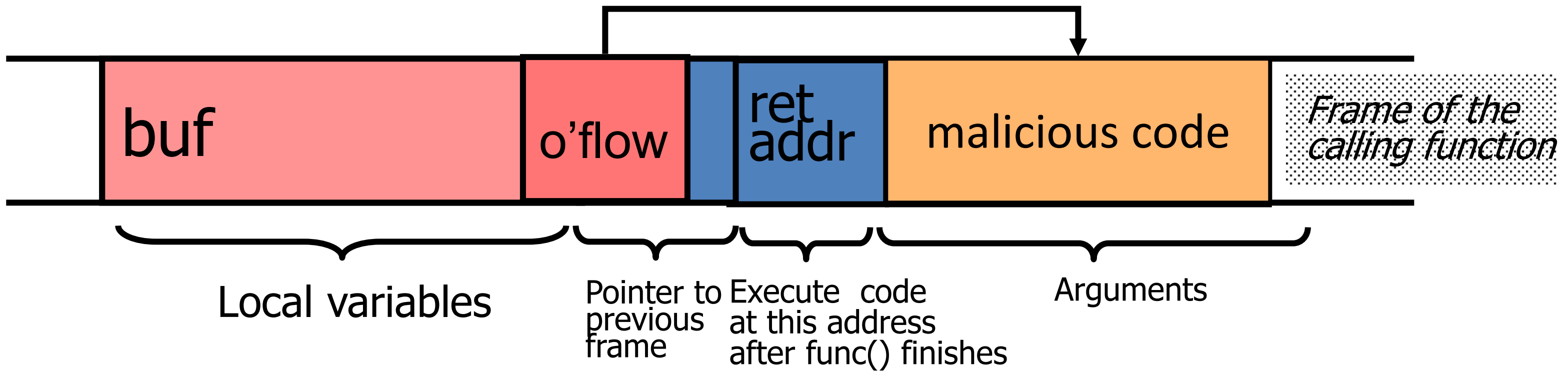
Send malicious code in s

Change the caller’s *saved frame ptr.*

- Pass more than 80 bytes in gets
- sfp now points to malicious code

Caller’s return address read from sfp

When func returns, mal. code runs !



Attack 5: Integer Overflow


```
static int getpeername1(p, uap, compat) {
// In FreeBSD kernel, retrieves address of peer to which a socket is connected
    …
    struct sockaddr *sa;
     …
    len = MIN(len, sa->sa_len);
    … copyout(sa, (caddr_t)uap->asa, (u_int)len);
    …
}
```

Checks that “len” is not too big

Negative “len” will always pass this check…

… interpreted as a huge unsigned integer here

Copies “len” bytes from kernel memory to user space

… will copy up to 4G of kernel memory

Time of Check to Time of Use

- Concurrency issue
  - –Successive instructions may not execute serially
  - –Other processes may be given control
- TOCTTOU: control is given to other process *between* access control check and access operation

TOCTTOU Example


```
int openfile(char *path) {
    struct stat s;
    if (stat(path, &s) < 0)
        return -1;
    if (!S_ISRREG(s.st_mode)) {
        error("only allowed to regular files");
        return -1;
    }
    return open(path, O_RDONLY);
}
```

Path to file

Extract file meta-data

Between check and open attacker can change path

- Initial path is regular file
- Later path is not

Adversary by-passes security

Open file

No symlink, directory, special file

TOCTTOU Defense

1. Ensure critical parameters are not exposed during pre-emption
   - openfile "owns" path
2. Ensure serial integrity
   - openfile is atomic
   - No pre-emption during its execution
3. Validate critical parameters
   - Compute checksum of path before pre-emption
   - Compare to checksum of path after …

Incomplete Mediation

- http://www.abc.com/subpage/userinput.asp?par1=(808)555-1212&par2=2011Sep10
- What if par2 is
  - 1800Jan01 (outside of range)
  - 2000Feb30 (non-existent)
  - 2048Min32 (undefined)
  - 1Aardvark2Many ?!?
- How to fix such errors ?
  - Have client-side code to verify input correctness
  - Restrict choices to only possible ones, e.g., drop-down menus …

Incomplete Mediation

- http://www.abc.com/subpage/userinput.asp?par1=(808)555-1212&par2=2011Sep10
- *Still vulnerable !*
  - The results of the verification are accessible in the URL
  - The (malicious) user can access and modify fields
  - Only then send to the server
  - The server cannot tell if URL came directly from the user browser or from malicious user

Use in Combination

- Can be used together
- Example: Attacker can
  - Use buffer overflow to disrupt code execution
  - Use TOCTTOU to add a new user to system
  - Use incomplete mediation to achieve privileged status
  - …

Firmware Supply Chains

- Are long and obscure
- Involve hundreds of modules (300+)
- Many tens of (sub)vendors
- Firmware is often flashed in factory (China)
- Relatively easy to compromise (<$100k)

Firmware

- Is critical
- Is completely overlooked
- Underpins everything on top

Firmware Compromise

- Is virtually **impossible to detect**
- Much easier than compromising foundries/chips
- Transforms the machine into an APT zombie
- Any “security” built on top is 100% compromised
- Almost the very definition of “sandcastle”

Modern Stack: Millions of Bugs. Literally.

Data	Application
VM OS Image	15,000,000 - 250,000,000
VM OS Kernel	2,000,000 - 28,000,000
Cloud Hypervisor	6,000,000
Motherboard BIOS/firmware	1,500,000
Motherboard IPMI/controller	2,000,000
Intel ME	100,000 - 200,000
Intel AMT	150,000 - 300,000
Intel microcode	
Cloud Network Fabric / SDN	500,000 - 750,000
Cloud Management Logic	14,000,000 – 25,000,000
Data Center Switching/Routing	

40,000,000-300,000,000 lines

2% bugs/line of code (BLOC)
600,000-6,000,000 bugs

5% actual exploits
30,000-300,000 viable exploits

Zero-day exploit
$500k+ on darknet

Exploit market
$150b

Private Machines
High Performance Zero Trust Infrastructure Tech

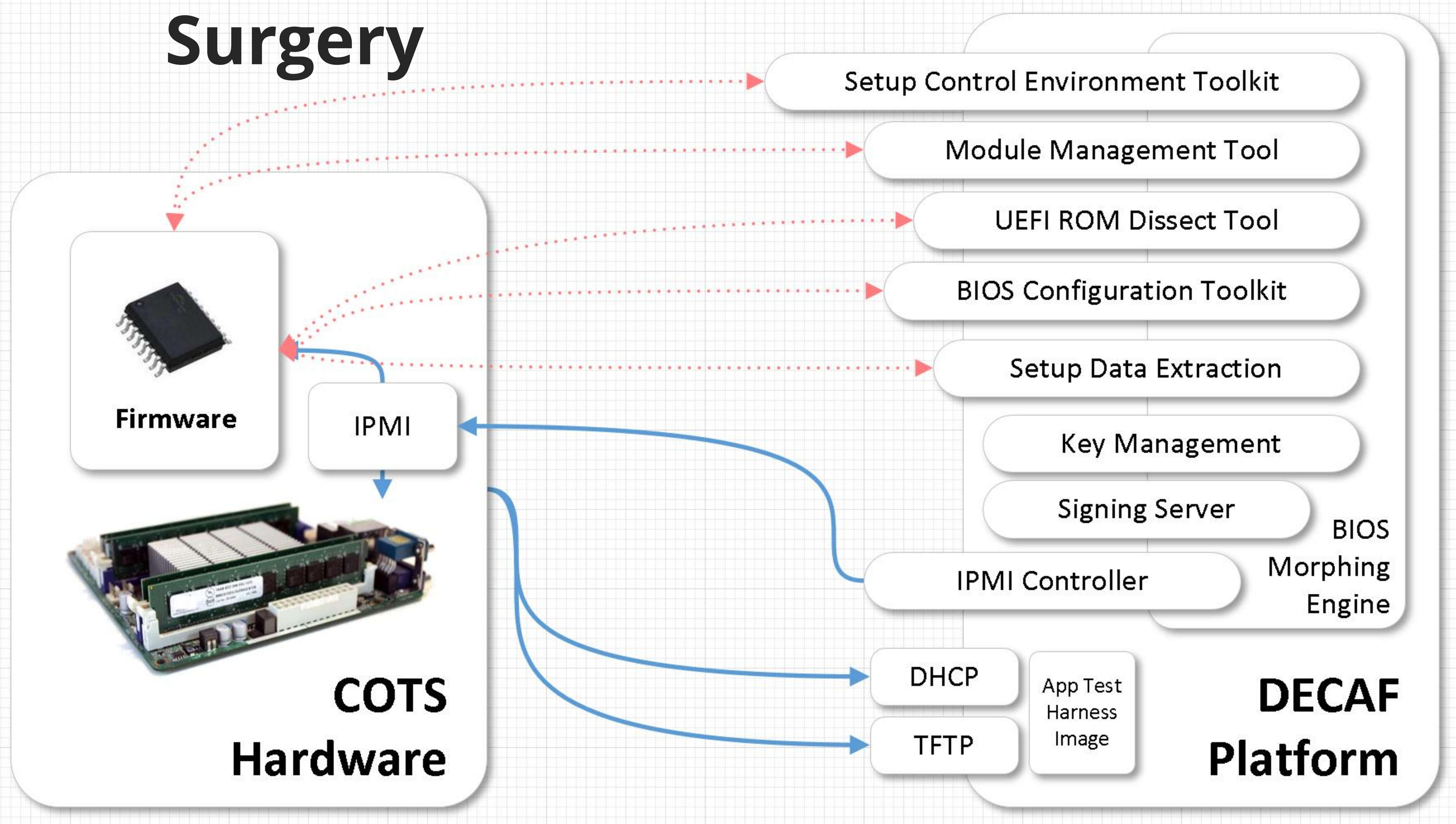
Firmware is impossible to fully secure

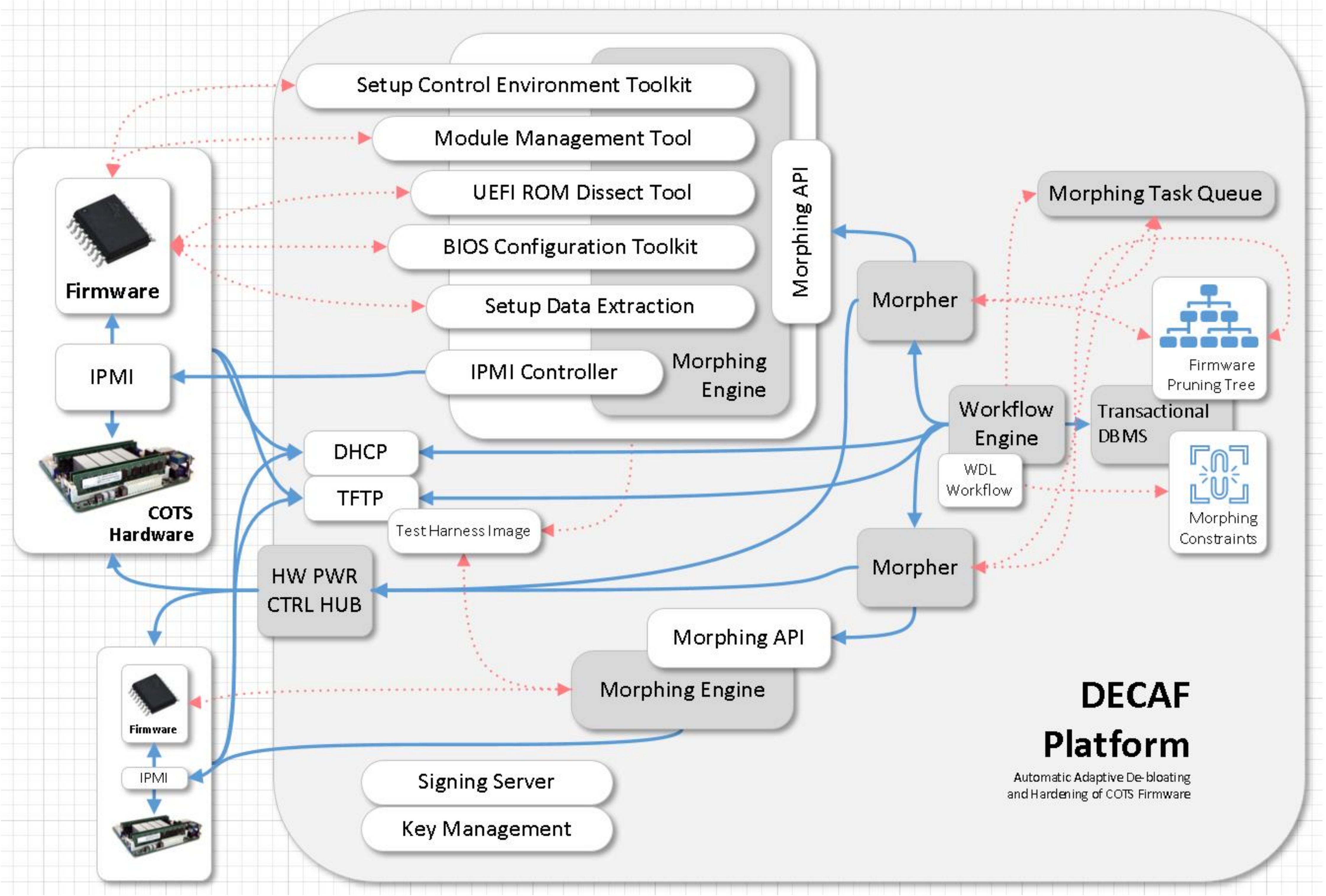
- Typical BIOS
  - 2-3m lines of code
  - about 60,000 bugs
  - about 3000 exploits
- Smallest custom embedded BIOSes
  - 20,000-100,000 lines of code
  - **at least** 400-2000 bugs
  - **at least** 20-100 exploits
  - not really usable in modern servers

80% of Firmware:

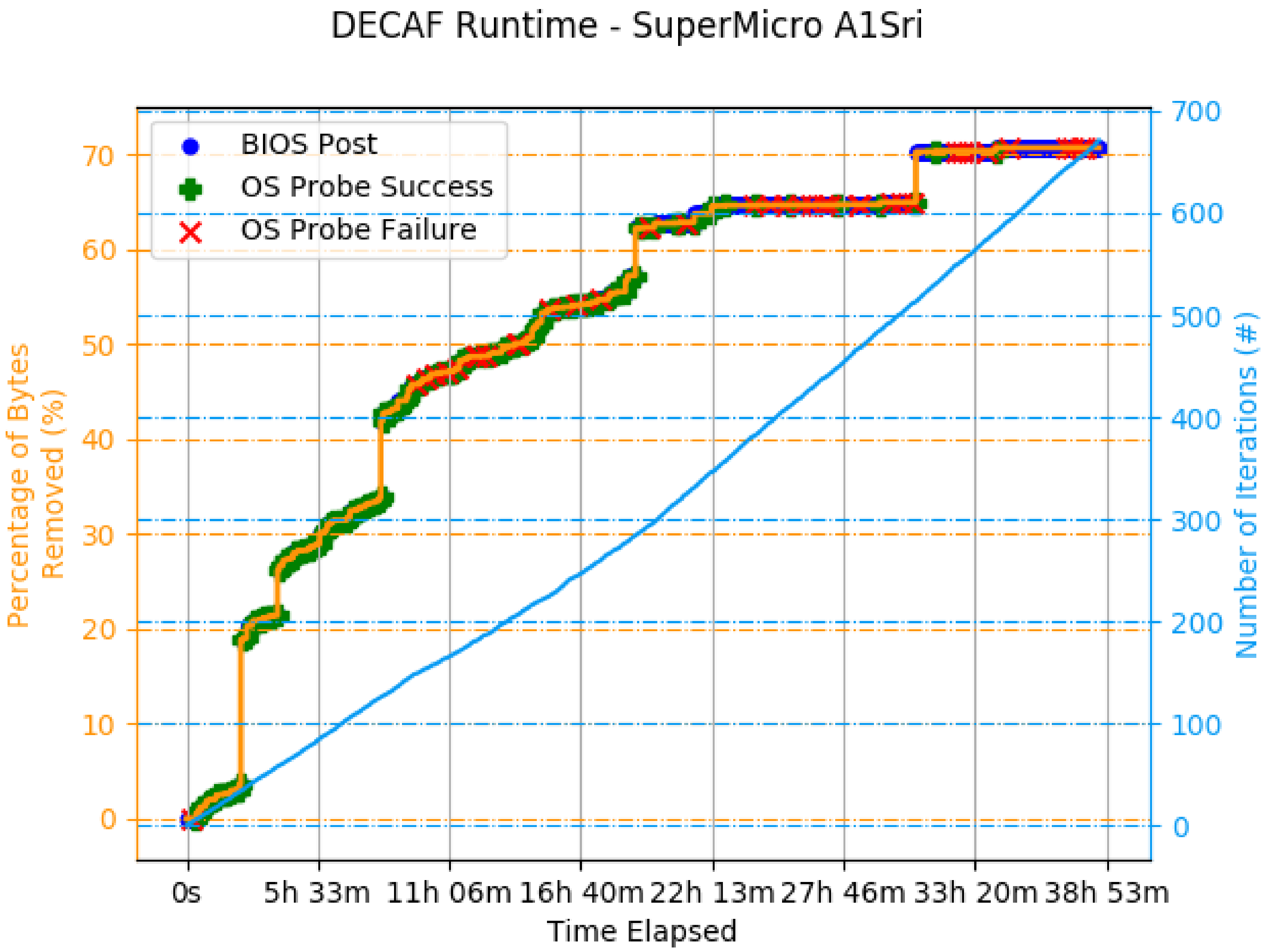
- is unnecessary
- is obsolete
- is full of bugs and exploits
- is difficult or impossible to update
- should be removed
- harden remaining core
- this can significantly disrupt supply chain attacks

Dynamic Firmware Surgery




Setup Control Environment Toolkit
Module Management Tool
UEFI ROM Dissect Tool
BIOS Configuration Toolkit
Setup Data Extraction
IPMI Controller
Morphing Engine
Morphing API
Firmware
IPMI
COTS Hardware
DHCP
TFTP
Test Harness Image
HW PWR CTRL HUB
Morpher
Workflow Engine
WDL Workflow
Transactional DBMS
Morphing Task Queue
Firmware Pruning Tree
Morphing Constraints
Morpher
Morphing API
Morphing Engine
Firmware
IPMI
Signing Server
Key Management
DECAF Platform
Automatic Adaptive De-bloating and Hardening of COTS Firmware

Results: SuperMicro

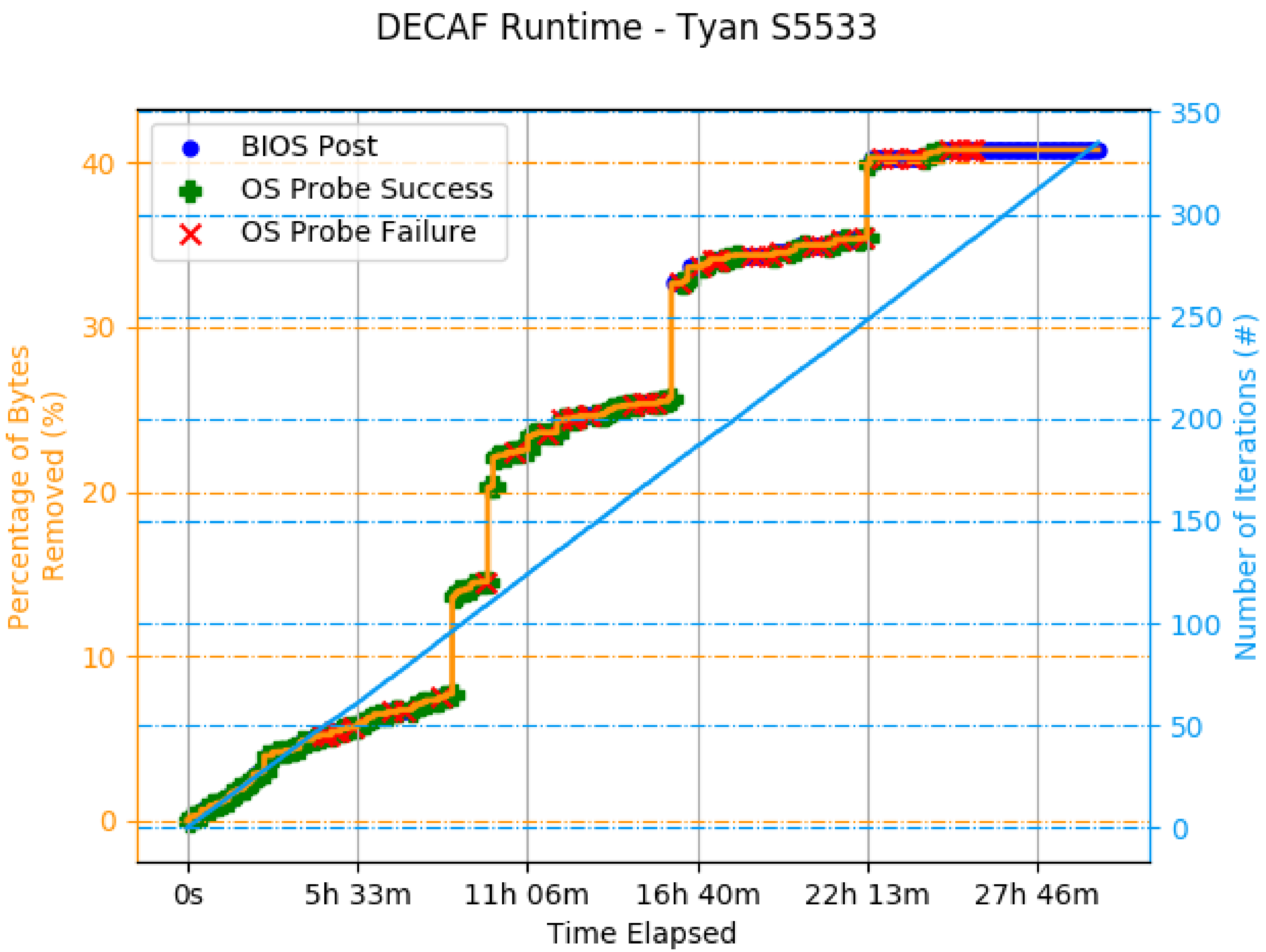


*Removed 152/244 modules

** ~62% of modules

** ~70% of binary

Results: Tyan



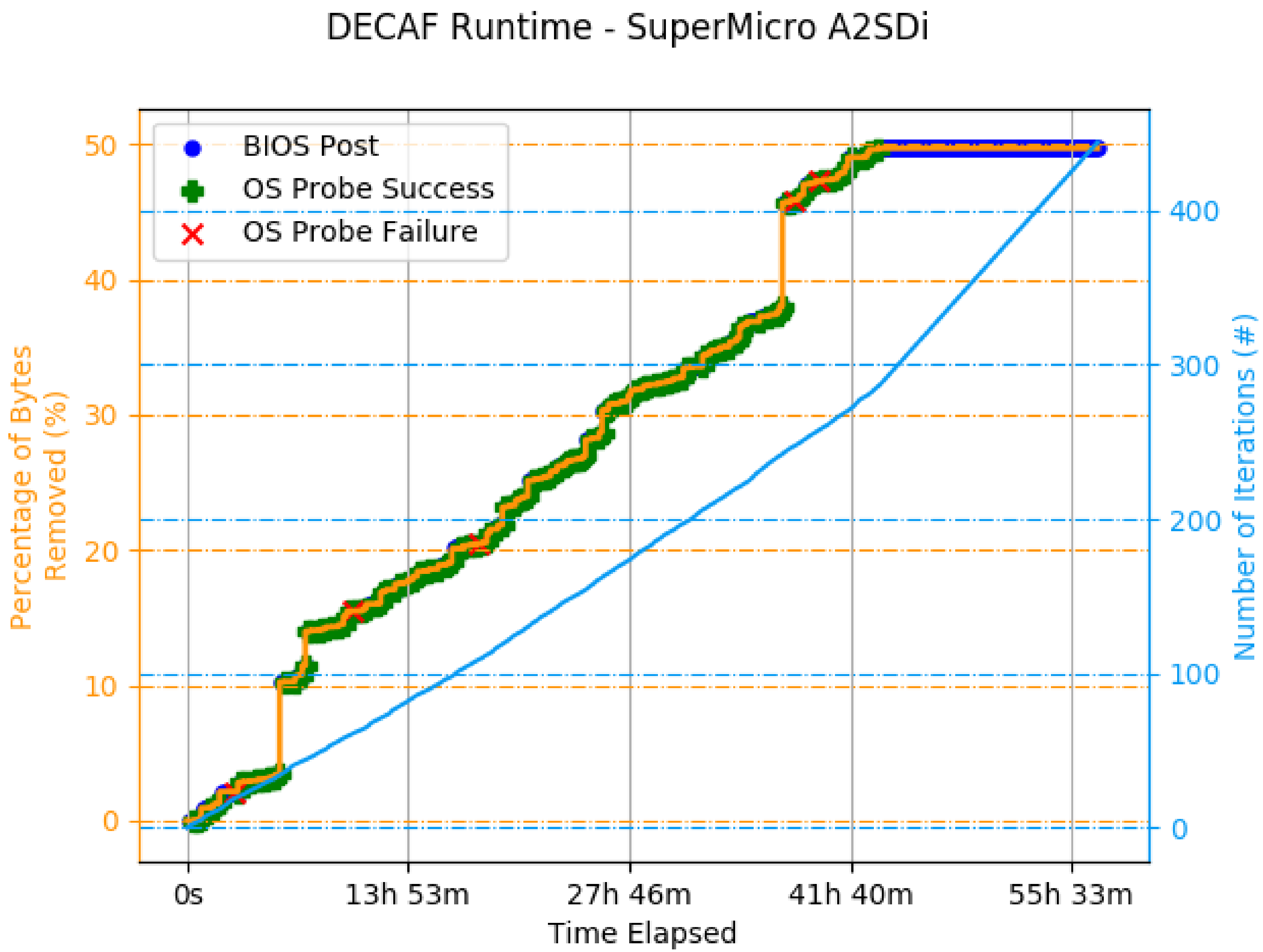
*Removed 134/194 modules

** ~70% of modules

** ~40% of binary

Private Machines
High Performance Zero Trust Infrastructure Tech

Results: SuperMicro



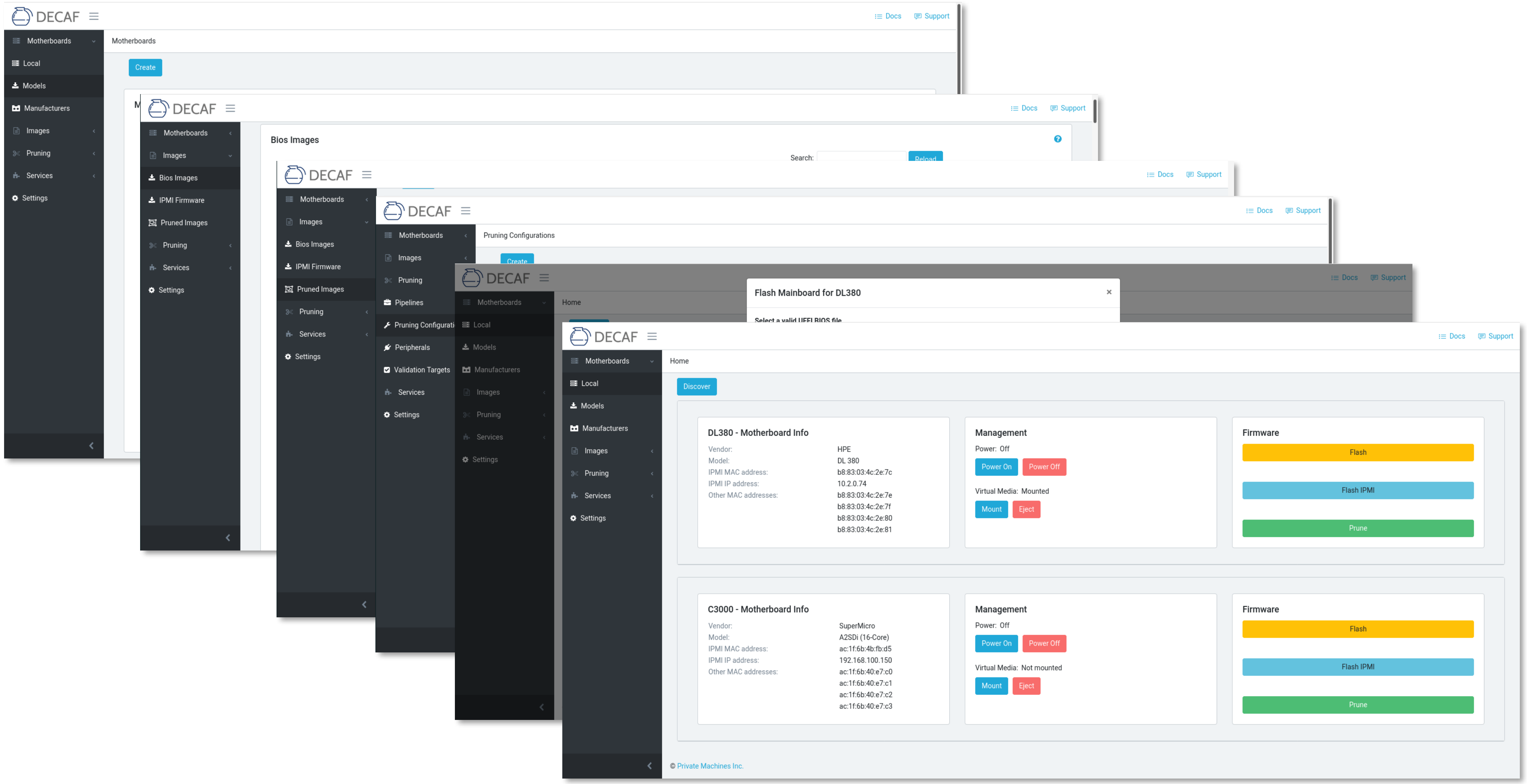
*Removed 154/312 modules

** ~50% of modules

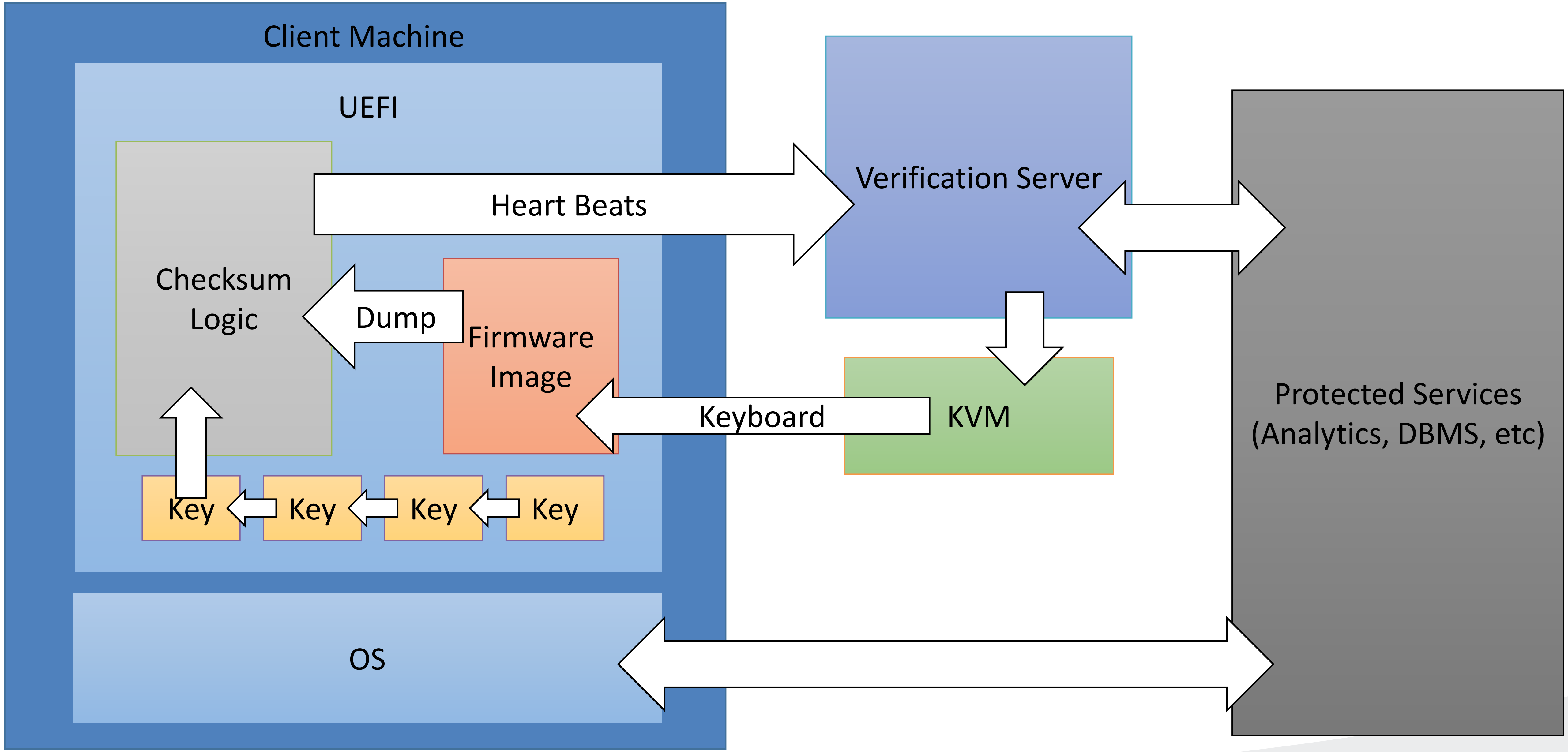
** ~50% of binary

Private Machines
High Performance Zero Trust Infrastructure Tech

DECAF In Action



DECAF DEFENDER



Take Home (Firmware)

- Firmware is critical yet very often overlooked
- Everything built on top depends on its security
- Even the best firmware has **thousands of bugs**
- Firmware **supply chains are difficult to trust**
- Reducing firmware vulnerability surfaces can significantly disrupt supply chain attacks