

CSE 227 Computer Security

Stefan Savage, Fall 2021

Low-level vulnerabilities and exploits

Two papers today

- Úlfar Erlingsson, Yves Younanm, Frank Piessens
Low-level Software Security by Example
Handbook of Information and Communications Security, 2010
- Sergey Bratus, Michael E. Locasto, Meredith L. Patterson, Len Sassaman, and Anna Shubina
Exploit Programming From Buffer Overflows to “Weird Machines” and Theory of Computation, USENIX ;login 2011
- Both are about the same topic, but described differently

Before we discuss papers

- Primer/review on low-level vulnerabilities and defenses
 - Largely covers content of 1st paper
 - I think this is the only class we'll do this, but necessary for those without security background

What is a software vulnerability?

- A bug in a software program that allows an unprivileged user capabilities that should be denied to them
- There are a lot of types of vulnerabilities, but among the most classic and important are vulnerabilities that violate “**control flow integrity**”
 - Translation: **lets attacker run code of their choosing on your computer**
- Typically, these involve violating *assumptions* of the programming language or its run-time system

Classic: Stack overflows

- Basic idea
 - Overwrite control data on stack to execute arbitrary instructions from input
- Canonical papers
 - AlephOne “Hacking the Stack for Fun and Profit”, Phrack 49, 1996
 - Dildog, “The Tao of Windows Buffer Overruns”, Cult of The Dead Cow cDC-351, 1998
- Best known wide exploit:
 - Robert T. Morris worm, 1988

Example 1: fingerd

- Spot the vulnerability
 - What does `gets()` do?
 - How many characters does it read in?
 - Who decides how much input to provide?
 - How large is `line[]`?
 - Implicit assumption about input length
 - What happens if, say 536, characters are provided as input?
- Source: fingerd code

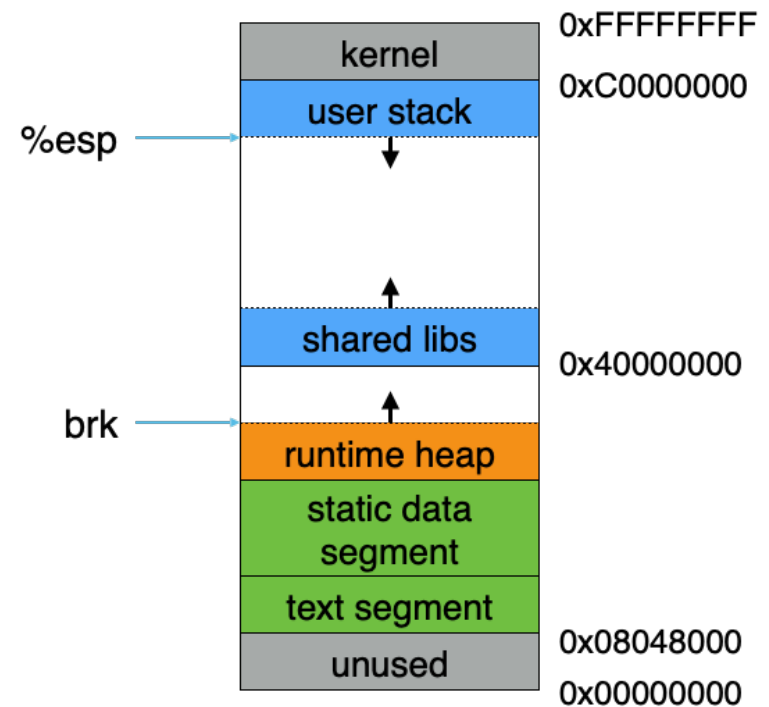
```
1 main(argc, argv)
2     char *argv[];
3 {
4     register char *sp;
5     char line[512];
6     struct sockaddr_in sin;
7     int i, p[2], pid, status;
8     FILE *fp;
9     char *av[4];
10
11     i = sizeof (sin);
12     if (getpeername(0, &sin, &i) < 0)
13         fatal(argv[0], "getpeername");
14     line[0] = '\0';
15     gets(line);
16     //...
17     return(0);
18 }
```

Some background/context

- How memory is laid out in a process
- How C arrays work
- How C function calls work

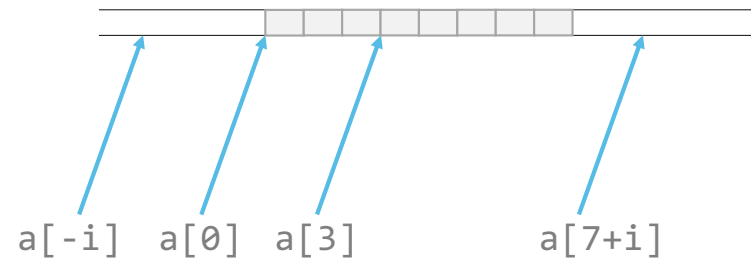
How process memory is laid out (Linux 32bit traditional, simplified)

- Stack
 - Locals, call stack
- Heap
 - i.e. malloc, new, etc...
- Data segment (globals, statics)
 - .data
 - .bss
- Text segment
 - Executable code



How do C arrays work?

- What's the abstraction?
- What's the reality?
 - What happens if you try to write past the end of an array in C/C++
 - What does the [spec](#) say?
 - What happens in most implementations?



Understanding Function Calls

- How does a function call work?

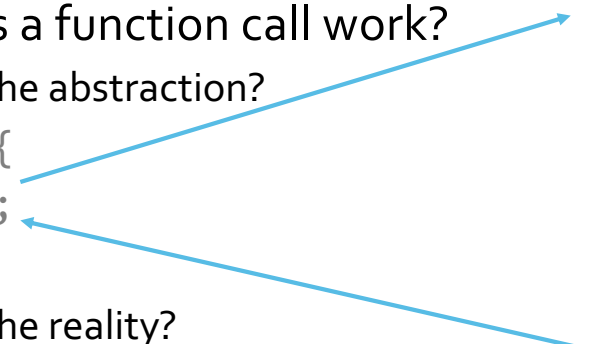
- What's the abstraction?

```
bar() {  
  foo();  
}
```

- What's the reality?

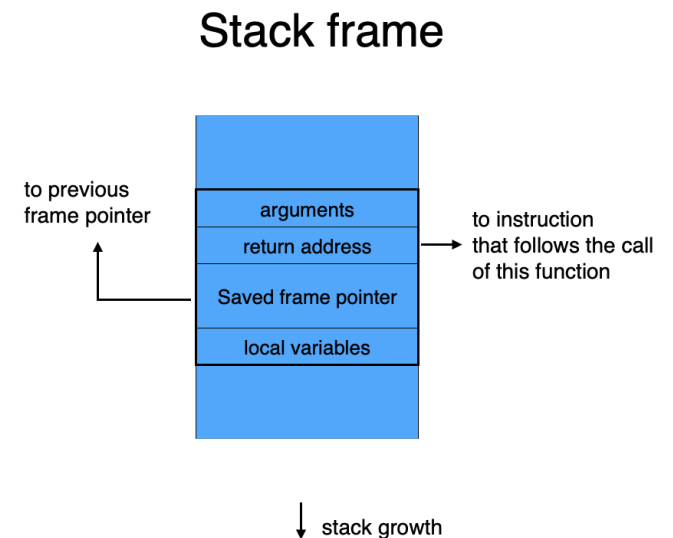
- Where does the memory for i from?
 - How does the called function know where to return to?

```
void foo()  
{  
  int i;  
  ...  
  i=20;  
  ...  
  return;  
}
```



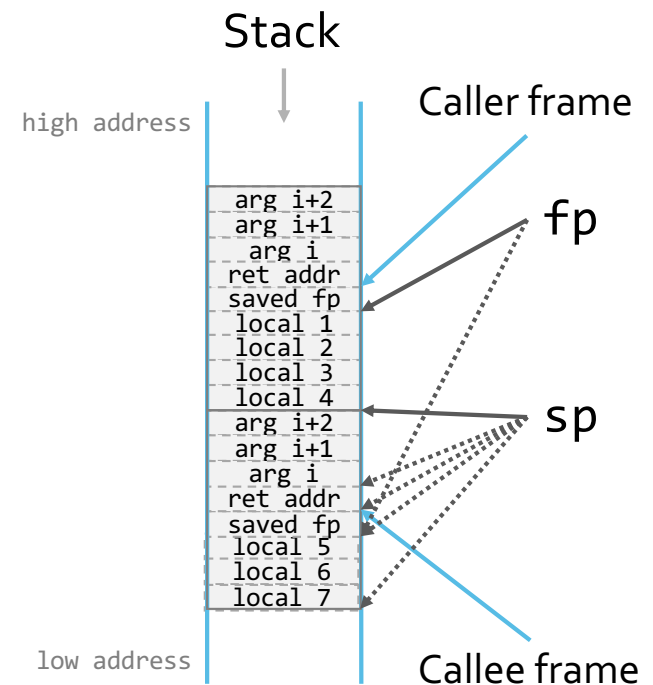
The Stack

- Stack divided into **frames**
 - Each frame stores locals and args to called functions
- **Stack pointer** points to the top of the stack
 - x86: stack grows down (from high to low addresses)
 - x86: stored in %esp register
- **Frame pointer** points to caller's frame on the stack
 - Also called (only by Intel) the *base pointer*
 - x86: Stored in %ebp register



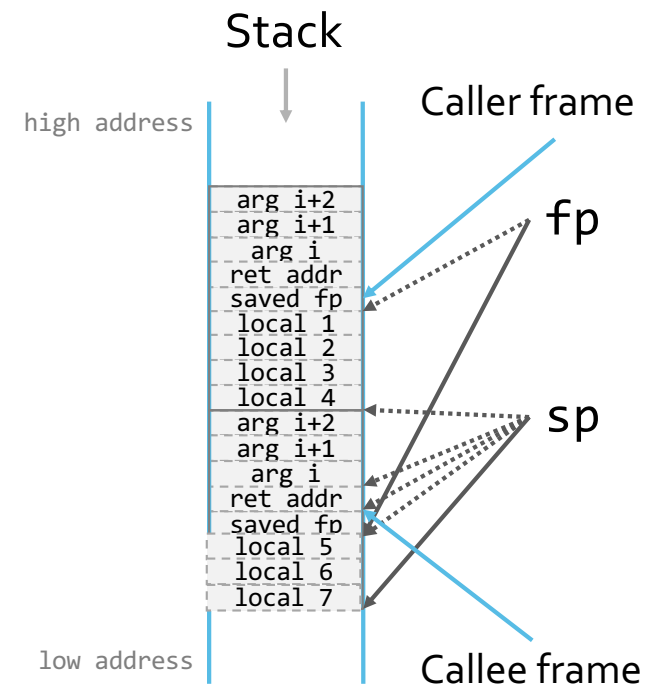
Understanding Function Calls

- Calling a function
 - Caller
 - Pass arguments
 - Call and save return address
 - Callee
 - Save old frame pointer
 - Set frame pointer = stack pointer
 - Allocate stack space for local storage
- Call Frame (Stack Frame)



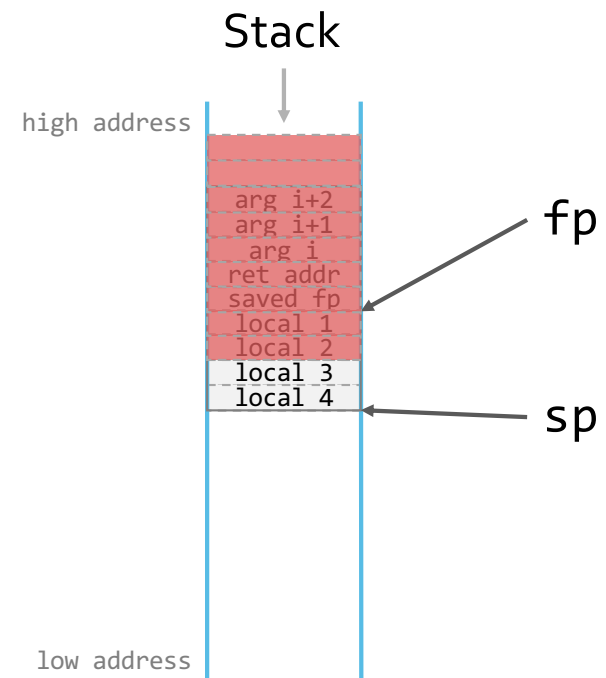
Understanding Function Calls

- When returning
 - Callee
 - Pop local storage
 - Set stack pointer = frame pointer
 - Pop frame pointer
 - Pop return address and return
 - Caller
 - Pop arguments



Smashing The Stack

- Mixing control and user data creates an opportunity for attackers
- What happens if you overwrite an attacker-supplied value past the bounds of a local variable?
 - Let's say we overflow `local 3`
- Overwriting
 - Another local variable
 - Saved frame pointer
 - **Return address**
 - Function arguments
 - Deeper stack frames
 - Overflow can happen *outside* of current function's frame
 - Exception control data



What are the key issues?

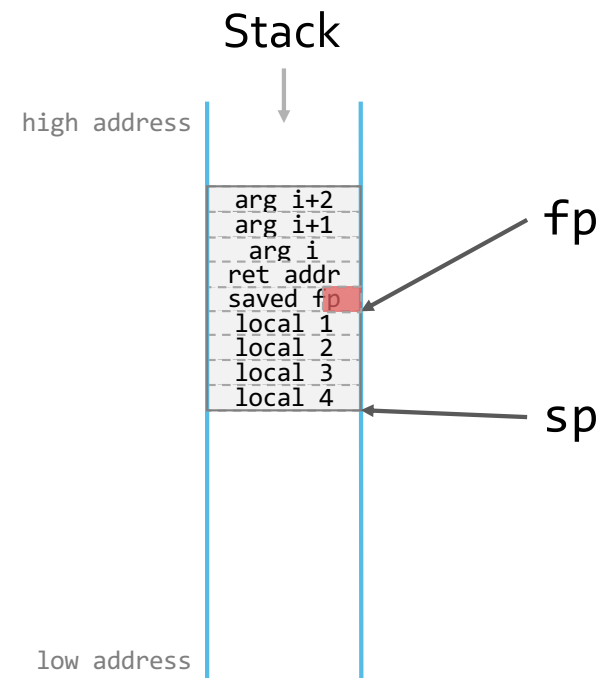
- The C language is weakly typed
 - Allows writing arbitrary values to arbitrary locations
- Control flow is dynamic, based on memory
 - Return address, function pointers, jump tables
 - If you overwrite these you can change control flow
- The processor doesn't know the difference between code and data
 - It will execute instructions from any location in memory

Vulnerabilities, threats and hindsight

- Just a bug or exploitable vulnerability?
- Lots of hot air expended on this topic
 - “Yes, you found a bug, but its not exploitable”
 - “This class of bugs is very hard to exploit”
 - “While the DoS threat is significant, this vulnerability can’ t be used for code injection”
- Historically these distinctions have changed with experience
 - Case in point: the off-by-one stack overflow
 - Historically, not considered a major control hijacking threat
 - Today, considered easy

Smashing The Stack: off by one errors

- But what if you can only overwrite **one word** or **one byte**?
 - Seems hard to exploit no?
- Overwriting the **saved frame pointer**
 - Upon function return, stack moves to an attacker-supplied address
 - Make up a **fake frame**, with return address of your choosing
 - When *that* function returns, its game over again
 - In general, control of the stack leads to control of execution
 - Even a single byte may be enough!

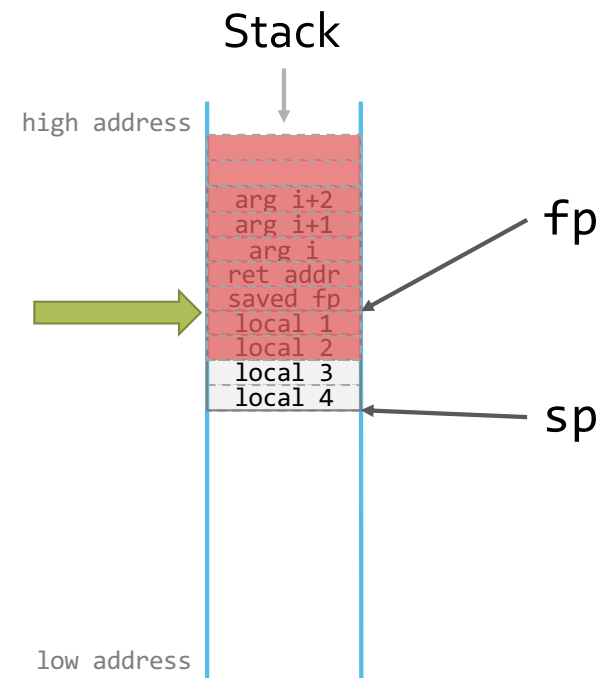


Defense/attack pattern

- All attacks exploit programming **assumptions** that are not **guaranteed**
 - E.g., that integers don't overflow, that inputs won't exceed the size of the allocated buffer, etc.
- Defenses frequently do the same thing
 - Assume attacker requires X to mount attack and then try to prevent them from getting X
 - But what if they don't really need X?
- Security literature is full of **attacks on defenses**

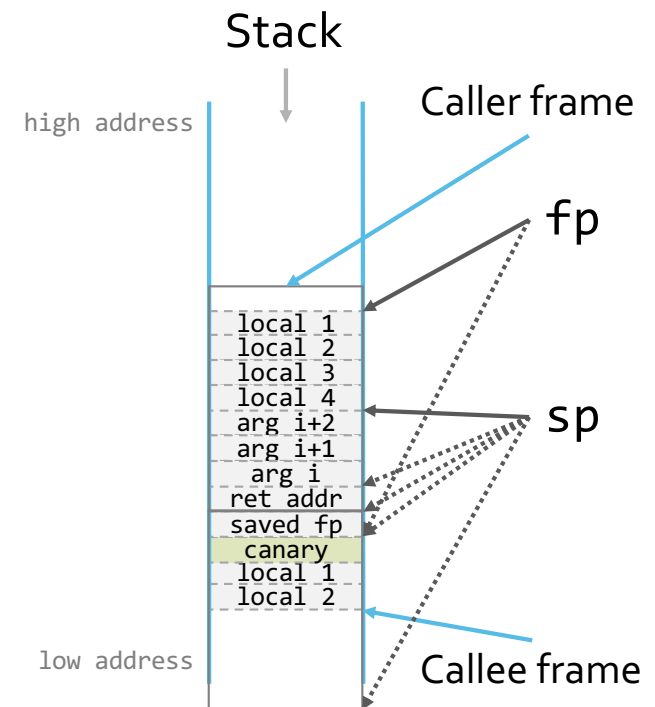
Stack Validation

- Stack-based buffer overflows
 - Typical attacks overflow local buffer into control data (i.e., ret)
- **Detect** overwriting of the return address
 - Place a special value (aka *canary* or cookie) between local variables and the saved frame pointer
 - Check that value before popping saved frame pointer and return address from the stack



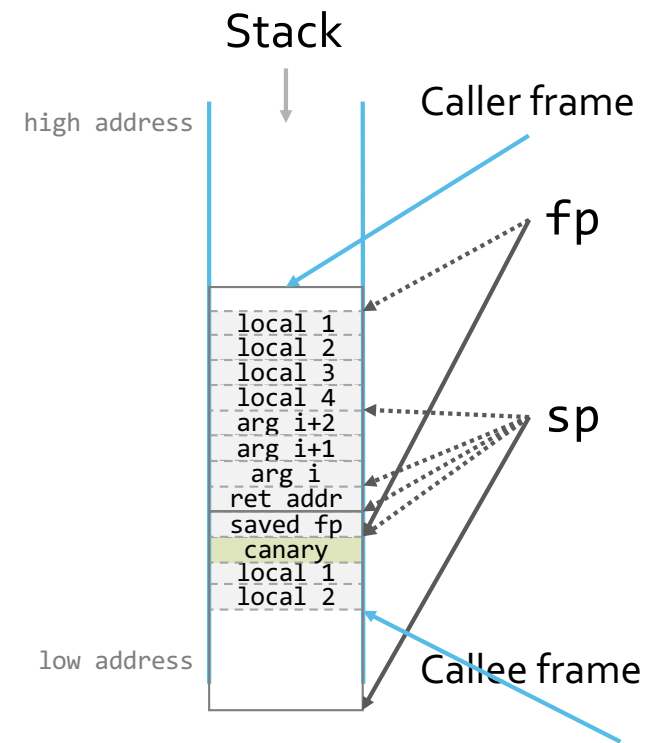
Stack Validation

- Calling a function
 - Caller
 - Pass arguments
 - Call and save return address
 - Callee
 - Save old frame pointer
 - Set frame pointer = stack pointer
 - **Allocate stack space for local storage + space for the canary**
 - **Push canary**



Stack Validation

- When returning
 - Callee
 - Check canary against a global 'gold' copy
 - Jump to exception handler if different
 - Pop local storage
 - Set stack pointer = frame pointer
 - Pop frame pointer
 - Pop return address and return
 - Caller
 - Pop arguments



Stack Canary Limitations

- How can stack canaries be bypassed?
 - **Assumption:** impossible to subvert control flow without corrupting the canary
 - Is it possible to overwrite the canary with a valid canary value?
 - Yes, if you can read or guess it
 - Can you cause trouble by overwriting non-protected data?
 - Yes, e.g., what about locals below the canary? (e.g., function ptr as a local?)
 - Is it possible to overwrite critical data without overwriting the canary?
 - Yes, stack overflow is only one technique... e.g., what if you can overwrite the address of a data pointer to point directly at the saved return address? Writes through that pointer will modify that return address, but canary is untouched.

Memory protection

- Use hardware to prevent instruction fetch from certain pages
- Idea: mark stack pages as non-executable
 - When processor tries to execute injected code in stack it traps
 - Fault handler stops program
 - Microsoft calls Data Execution Prevention (DEP)
 - Also can do mark heap the same way (issues?)
- **Assumption:** need to inject new code (i.e., shellcode) on the stack
- True?
 - No... can point to existing code (e.g., libc system() function, aka return-to-libc)

Advanced technique: Return-oriented Programming

- Malicious code assumption
 - If I can prevent new malicious code from being introduced or executed, then I'm fine
- Assumption turns out to be wrong
 - Malicious code is a subset of malicious computation
 - Ret-to-libc attacks are very simple example
 - No malicious code executed!
 - Turns out it can be generalized....

Thought experiment

- Suppose you have a stack overflow but can *only redirect control flow to existing code*
 - You can still jump to any legitimate instruction
- What if you jump into the middle of some code and that code ends with a RET instruction?
 - Where does control flow go now?
 - The return address pointed to by the stack pointer
 - Who controls that value?
 - The attacker does (because they had an overflow)
 - The stack pointer increments; repeat

Return Oriented Programming (ROP)

(Hovav Shacham)

- ROP idea: make shellcode out of **existing application code**.
- Stitching together arbitrary programs out of code gadgets already present in the target binary
 - **ROP Gadgets**: code sequences ending in ret instruction.
 - Commonly added by compiler (at end of function)
 - But **also** (on x86) any sequence in executable memory ending in `0xC3` (`ret`).
 - x86 has variable-length instructions
 - Misalignment (jumping into the middle of a longer instruction) can produce new, unintended, code sequences
- Overwrite saved return address on stack to point to first gadget, the following word to point to second gadget, etc
 - Like a weird virtual instruction set...
- Stack pointer is the new instruction pointer in this crazy world

Return Oriented Programming (ROP)

(Hovav Shacham)

- It turns out you can use these sequences to build a complete “virtual instruction set”
- Can build arbitrary new bad programs that are made completely out of “known good” instructions
- Can be largely automated
 - 13 years ago, two students in 227 built the first compiler for ROP
- Idea is super-common in modern exploits (although rarely using full generality)

CFI: Control Flow Integrity

- Detection regime
- We'll cover more later, but basic idea:
 - Check, at run-time, that execution path is allowed by original program
 - Insert “tags” in code before each branch target and when branching, first check that target's tag matches expectation
 - Like stack canaries, but for control flow

Address Space Layout Randomization

- Key idea: randomize code/data layout to minimize exploitable invariants
 - Example: randomize base address of stack, heap
 - Some trickiness:
 - Limitations on where objects can be located
 - Hard to re-randomize once a program is running (e.g., server programs that fork())
 - **Assumption**: attacker doesn't know where to transfer control to
- True?
 - Derandomizing: address leaks via interfaces, search using crashes as Oracle
 - Heap-spray: violate assumption that you need to know address of target code

Ok, back to our papers...

- Úlfar Erlingsson, Yves Younanm, Frank Piessens
Low-level Software Security by Example
Handbook of Information and Communications Security, 2010
- Sergey Bratus, Michael E. Locasto, Meredith L. Patterson, Len Sassaman, and Anna Shubina
Exploit Programming From Buffer Overflows to “Weird Machines” and Theory of Computation, USENIX ;login 2011
- Both are about the same topic, but described differently

Meta-questions

- Who are the authors?
- Why are they writing the paper?
- Why are they publishing it where they are?

Weird machines

- What is a “weird machine”?

For next time...

- Vulnerability finding techniques
 - EXE: Automatically Generating Inputs of Death, CCS 'o6
 - Evaluating Fuzz Testing, CCS '18