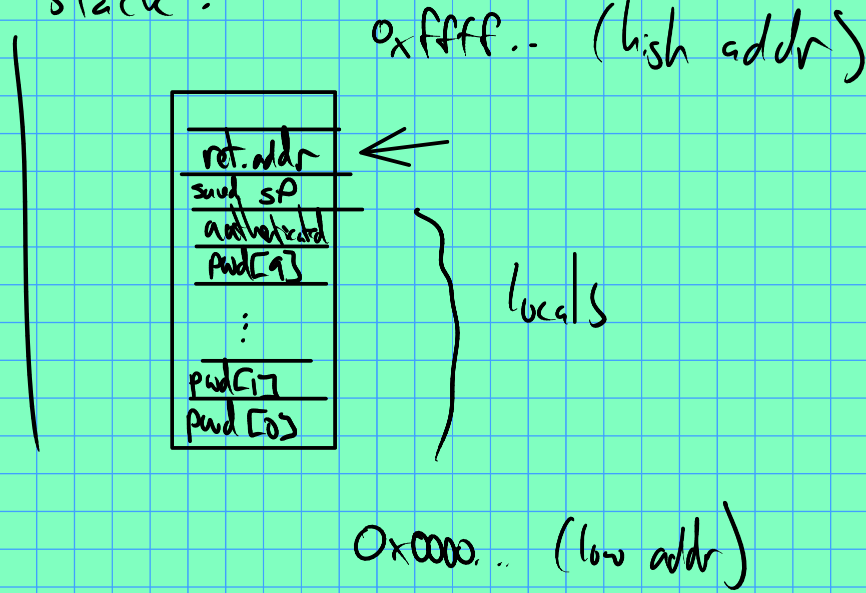
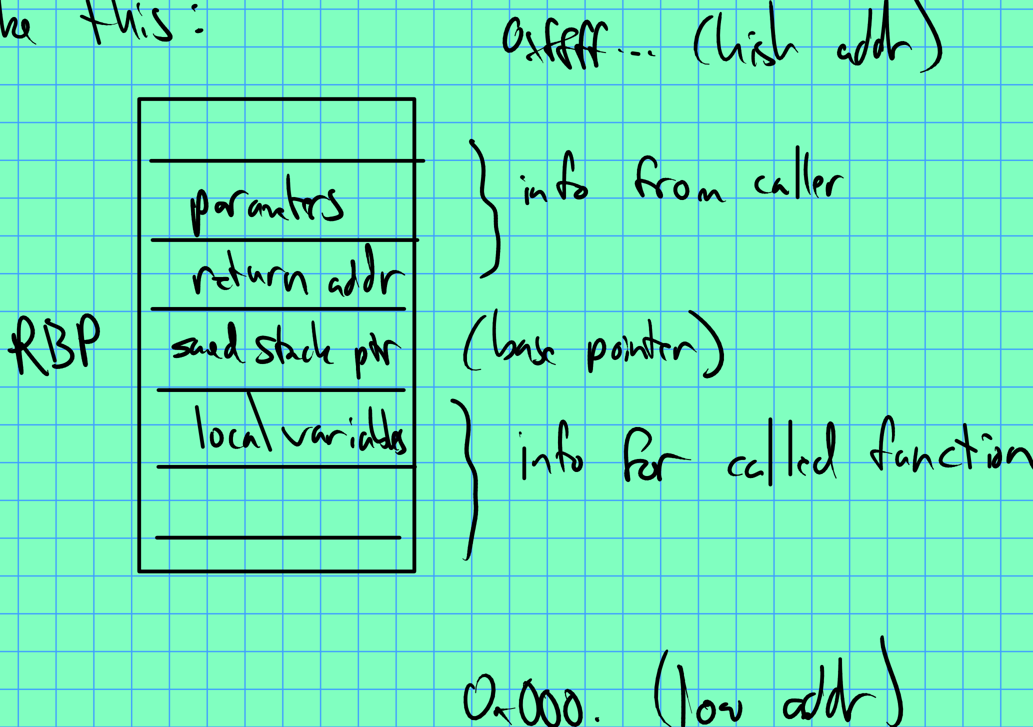
Software security: control-flow integrity.

local variables on call stack:

```
int authenticated = 0;
char pwd[10];
```



Quick review on call stack. For x86_64, looks something like this:



observations: call stack is fragile... by manipulating return address, we can change what instructions a program performs!

old school "stack smashing"

Setup: ① victim program uses a fixed length array on its stack to hold user input.

② user input is read into array <u>with no bounds checks</u> (e.g. via scanf("%s"))

prepare attack string like this:

$$\text{"}\underbrace{\sim\sim\sim}_{\text{fits in buffer}}\ \underline{\text{modified ret addr}}\ \underline{\text{binary attack code} \ldots\ldots}\text{"}$$

Hurdles for attacker:

proper return address seems hard to guess...
however, one can improve the odds of success using a "NOP-slide"

$$\underbrace{\text{NOP NOP NOP} \ldots\ldots \text{NOP}}\ \langle\text{Nasty code}\ldots\rangle$$

if we guess anything in this range, we win!

(Note: NOP can be useful, e.g. for aligning code on "nice" (16 N) byte boundaries...)

Other issues: might have to be creative to avoid \x00 (null characters) in your attack code!

What can we do about it??

— write better code! avoid strcpy and scanf on untrusted inputs.
(Static analysis is not bad in recent gcc.)

— W ⊕ X ("write exclusive-or execute")

Mark call stack as non-executable.

(There's hardware support for this since... a while)

(observation: attack string would have been on the call stack...)

Adaptation from attackers: return to libc instead...

(Note: libc has tons of potentially useful stuff like exec(...))

addresses of libc functions were actually easy to guess!

Defense to return to libc: ASLR

(Address Space Layout Randomization)

Idea: each process will have library func, stack, and heap in random addresses.

On x86-64, seems to be pretty good entropy!

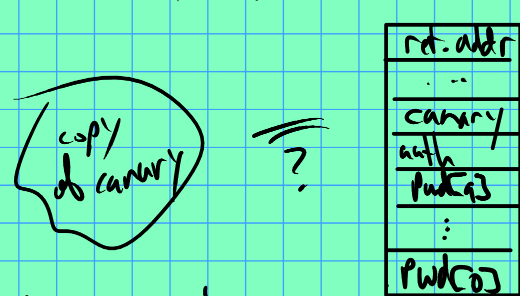
$\approx \log_2 1000000$ bits ?

— Stack Canaries

Idea: store single random value in 2 locations:

one on stack between locals + ret. addr;

the other copy is off the stack.



Now add to function epilogue: if (stack canary ≠ copy)

exit(1);

Pretty effective!

More sources of vulnerability: printf.

Issue: try to avoid ever using it w/ only one (non-literal) argument!

good: `printf("%s", buffer);`

bad: `printf(buffer);`

dangerous if user supplies buffer!

Say buffer = "%s%s%s"...

This allows one to print data from the stack.

Worse: could actually print memory from an address of the attackers' choice!

Key observation: nasty format string also stored on the call stack and printf is pulling arguments from the call stack:

`"\x7f\xff\xab\x01 %i %i %i %i %i %s"`

address you want to read

annoying to get right # of %i's ...

%s will print what is @ memory location

Scarier still: %n. Could write to address !

Further, this could be done without touching any canaries...

Return Oriented Programming (ROP)

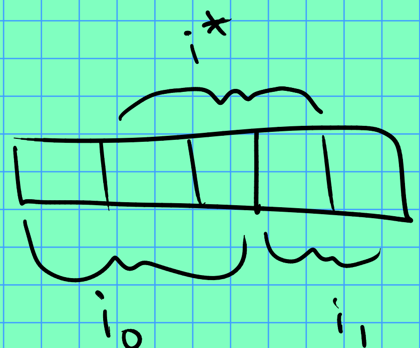
Return to libc has some limitations:

— can only call libc functions, might be harder to execute arbitrary code.

Idea: look for useful snippets of assembly (living inside libc code) which end in a 'ret' statement.

Set up memory + registers... then 'ret' to next useful snippet...

Added benefit: can even execute instructions not found in libc!



accidentally produced i^* as well!

produced by compiler