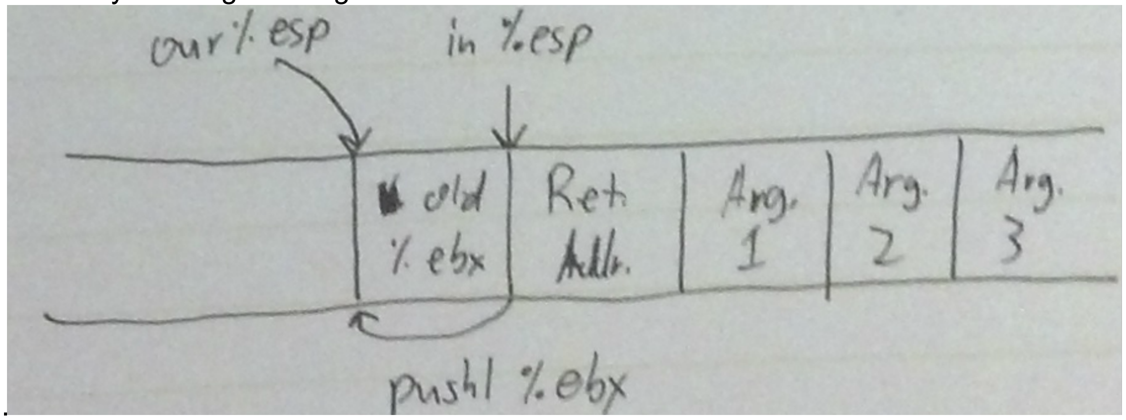
- Wc is a better linecount
- But weird boom function that prints boom
- Figure out an input to this program that will cause it to print boom
- In order to figure out this input, we need to reason about exactly what is on the stack

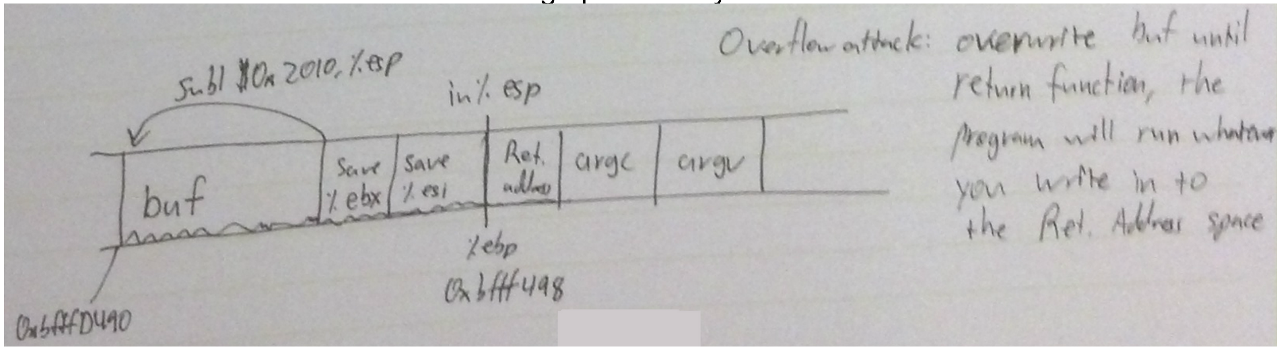
Items in memory reacting to being pushl'd.



Stack-smashing:

- %ebp is a register to keep track of arguments for use in larger functions (%esp moves)
    - Called the 'Frame Pointer'
    - The book says that %ebp is a mandatory part of the calling convention and
    - The book says that arguments are always referred to by the frame pointer -- the BOOK LIES!
    - The mandatory part is that it must be restored
    - %epb USUALLY refers to arguments

How we smash-attack the function to bring up hello kitty.



subl $0x2010, %esp
in % esp
buf
Save % ebx
Save % esi
Ret. addr
argc
argv
%ebp
0xbfff498
0xbfffD490
Overflow attack: overwrite buf until return function, the program will run whatever you write in to the Ret. Address space

New GDB Commands

- display/10i $pc == Display the next ten commands
- p $ebp = "print %ebp" → registers use $ at the start.

Verbose Notes

1. Today we will be going through more example files of assembly code. All of this code is available in the class code repository. Here are details about each of the example files:
    a. File F37
        i. What does this do
            1. Comparing two arguments
        ii. What are we returning?
            1. Some jump but we don't know what
            2. In order to unpack this need to unpack some of the weirder aspects
        iii. Actual code
            1. Returning the max of a and b
        iv. Why does this compile into what it does?
            1. Cmpl
            2. Jge
                a. Jump greater than or equal to
                b. But what is greater than or equal to
                c. Only has one argument, and that is .L2
                d. Looks like location in the program
                e. So it is saying that if it is greater than of equal to something else
        v. Let's try to peel this backwards to figure out what is in the registers
            1. We know we have the function:

```
f(int a , int b)
    if (a>b)
        ret a
    else
        ret b
```

1. Which register holds a?
    a. Ecx

        b. A -> %ecx
        c. B-> %edx
        d. Where is the return value placed?
            i. %eax
ii. So what is jge testing?
    1. Start with what we know, and work forward and backwards
        a. Do values in ecx and edx change?
        b. Ecx never changes
        c. And we know the return value goes in eax
        d. So know we know what the jge is comparing
            i. What do we return if the jge fails
                1. We move ecs into eax and fall through
                2. If it fails, we return ‘a’
                3. So therefore what is jge comparing?
                    a. Comparing b >= a
                    b. So jge is asking IS b >= a
iii. So now the missing piece
    1. Conditional branches in x86 work on a HIDDEN REGISTER, which is called %eflags
        a. Generic term for registers like this is condition code
        b. Almost every arithmetic expression set the condition codes as a side effect
            i. Add does not just add, it also sets condition code
            ii. Only one that does not is lea
    2. C.Codes
        a. Set as a side effect of every arithmetic instruction
        b. Flags
            i. ZF – Last # was zero
            ii. SF – Last # was negative (31st bit was 1)
            iii. CF – Unsigned overflow
            iv. OF – Signed overflow
        c. What does jge test?
            i. !(SF ^ OR)
            ii. Let’s compare with unsigned version of comparison (a stands for above)
                1. Jae
                    a. !CF
                    b. A lot of arithmetic has the same meaning in 32it whether or not it is signed
                    c. But comparison not this way
                2. Is 0x80000000<1?
                    a. Not if unsigned
                    b. But if signed, the first is negative, so then it is true
                    c. So comparison has two varieties, signed and unsigned

                d. Addition, subtraction, all of these are the same for signed unsigned
                e. But comparison depends on the sign of operands.
        3. Jge has no arguments, so what are the arguments to the cmpl that set the flags?
            a. A and b
            b. What arithmetic is cmpl doing?
                i. Subtraction, b – a
                ii. This is the magic of the cmpl instruction
            c. Cmpl R1 R2
                i. Like doing subtraction and throwing result away, all cmpl does is set the c.codes
        4. Now let's reason through some examples
            a. Zf
                i. Flags are always set
                ii. So other flags (SF CF, and OF) are all set to false
                iii. Signed overflow means that the answer has a different sign then you would expect
                    1. For example (-1) – 1 should be negative, but if because of wrap around it is positive, a signed overflow has happened
            b. cmpl $2, $-1
                i. -1 – 2 = -3
                ii. !ZF, SF, !CF, !OF
                iii. Is negative one less than or equal to 2, yes
                iv. So this is like -1 -2 = -3
                v. Because -1 is not equal to 2, so jge is false
            c. Sometimes the previous instruction is add or multiple, or a comp, but a comp instruction is usually what it done, because all it does is set flags

b. F38
    i. What does this do?
        1. Same
    ii. What it was intended to do, was to show that compiler could do different code
    iii. Could have done this code in a different way, with a jl instruction
    iv. By switching the ordering of the branches, could get equivalent assembly for the input code, and what the compiler generates is based on its estimation of what is likely
    v. The way modern processors got fast, is that they guess what they have to do in the future, and start working on it now
        1. This is called pipelining
        2. Modern processors enjoy long straight lines of instructions without jumps or memory reads

3. When there is a branch in the control flow, the processor has to guess which way it will go
    a. Processors have to guess fast and correctly, and avoid hiccups if the guess is wrong

vi. So what does this assembly do (particularly the many "nop"s at the end)

```
Movl
Movl
Movl
Cmpl
……
nop
nop
nop
nop
…
```

i. What does nop do?
1. No operations
2. Same in terms of function as previous
3. But this code is getting faster in one of the cases
    a. This is likely to be faster when a > b
    b. Why?
        i. If a > b, which branch is taken? The false branch
            1. Because a is in eax and that is what we return
            2. Where is the false branch located?--close to start
            3. The true branch is father away
            4. If the true branch is followed, have to load them in because they are farther away
            5. Compilers actually do this all the time, based on which branch they think is more likely
            6. Get benefits of 20%, 30%, 40%, if the compilers smash the likely cases together in memory, and put the other ones somewhere else

ii. What does __builtin_expect((x),0) do?
1. Return x, but behave as if x is likely to return 0

iii. What does Likely(x) __builtin_expect((x),1) do?
1. Return x, but likely 1

iv. Another way to accomplish performance gains this way is to use profile info
1. Compiler can log which branches were taken, run the program many times, average results, and feed back in the appropriate information

b. F39

i. Eax is the return value loaded with a, edx loaded with b
ii. Cmpl of x, %edg
iii. Je is equal
iv. So if jge means last thing >=0
v. Je means jump if the last thing was 0

            1. Cmpl means second argument being compared to some global x
            2. If b == x return a, otherwise return b
c. F40
    i. Same as previous, but not equal instead of equal
d. F41
    i. Jae
        1. Unsigned
        2. If a>b, but using unsigned integers
e. F42
    i. Testl
        1. This does a similar thing as compl
        2. Compl is subtraction and then throw away the answer
        3. Testl is "bitwise &" and then throw away the answer
        4. For example: if a & b return a, otherwise return b
f. F43
    i. Testl %eax, %eax
g. F44
    i. We have two arguments, but what are the types of those arguments?
    ii. Sure that edx is a pointer
        1. Because inside parenthesis
        2. So lets say the two arguments are p and b
        3. If p is not null, then return *p
            a. Otherwise return b
            b. Compiler has preloaded the eax register
            c. If it is 0 then return immediately
            d. If the p argument is non zero, fall through and dereference that argument and put it into the register
            e. So the function is

```
Unsigned f(unsigned *p, unsigned b)
if(p)
        return *p
else
        return b.
```

a. F45
    i. How do we know this is a loop?
        1. One magic property-backwards jump—always had by loops
        2. At the end of the .L3 block
        3. Figure 2 (see above)
            a. Basic block control flow graph
                i. Basic block is a sequence of instructions that are always executed as a unit
                ii. Starts at beginning and goes through the end
                iii. Might have many ways to exit, might jump to another function, but every route that enters the basic block enters through the basic block

        ii. What does this function do?
            1. Returns a + a + 1 / 2
            2. Moves argument
            3. Increments the counter, and then adds the counter to the return?
            4. Once the counter gets to the end value it exists
            5. Rep ret
                a. For amd processors, a little different from other processors
        iii. Look for counter—it's the register begin compared at the end of the loop
        iv. Here, that comparison is edx
    b. F46
        i. Is there a loop?
            1. Yes
        ii. Where is the loop?
            1. Jne .L8
            2. Loop is blocks from L8 to L3
        iii. How many arguments are there here?
        iv. Why are we putting some garbage onto the stack, and then carefully taking it off the stack at the end of the function
        v. Calling convention
            1. Distinction between registers in x86 that every function is guaranteed to put back to the original value, and other registers that a function can use however it want
                a. So there are 'scratch' registers the function can use, and precious registers the function must restore
            2. Scratch registers are called "CALLER-SAVED"
            3. Restore on exit registers are called "CALLEE-SAVED"
            4. If a function wants to use one of these registers, has to save it's value on entry and restore on exist
            5. Is eax scratch or restore?
                a. CALLER-SAVED
                b. The other CALLER-SAVED registers are ecx and edx
            6. In all of the examples so far a lot of eax ecx and edx
            7. The compile is using those registers because it can use them for free
            8. A function can make the assembly code shorter if it does not have to save any info from the registers and then restore
            9. Caller-saved
                a. Esp ebp ebx esi edi
        vi. What is returnedin this file? Eax
        vii. What else is eax?
            1. The loop variable that is changed every time it loops
            2. Because increment every time after L3 and then to see
        viii. Important point is that you can take this apart relatively easily and see what you are doing
    c. F51
        i. Loop with multiply
        ii. Factorial function

d. Return to the hello kitty program with a secret malicious function
    i. How do problems like this happen
    ii. Here we have a program, smash01.c
    iii. What does this program do?
        1. 'gets' just reads lines, and for every line just does ++lines and then prints the numbers of lines in standard input
    iv. (Wc is a better line count program)
    v. But there is a weird boom function that prints boom
    vi. Let's figure out an input to this program that will cause it to print boom
    vii. In order to figure out this input, we need to reason about exactly what is on the stack
    viii. How are buf and lines arranged?
    ix. Lets try it out under gdp
    x. Gb smash01
        1. b main (set breakpoint)
        2. display/10i $pc (display next ten assembly instructions)
        3. r (run program)
        4. Ignore instruction we don't understand
            a. Push %esi and push #ebx
                i. Saving CALLEE-SAVED registers
                ii. The functions we have been looking at earlier are so small, it has optimized away this part
                iii. Uses ebp instead of esp to refer to arguments
                iv. Ebp used to refer to arguments in large and complex functions
                v. ebp often refers to arguments
        5. p $ebp
            a. It is zero
            b. So push ebp, setup ebp and push and save more variables, save esi and save ebx
        6. Now there is something weird to align the stack to 16 byte boundaries
        7. And then add a lot of spaces
            a. Sub $0x2010,%esp
        8. So the first argument has 8 off of ebp (one of the reasons we use ebp it that is makes offsets smaller)
    xi. In the pset, probably will not use p a lot
            a. No debugging info
        2. P &buf[0]
            a. 0xbffd490
        3. p $esp
            a. oxbffd480
        4. Base pointer p $ebp
            a. 0xbffff498
        5. The gets function doesn't know how big the buffer is
            a. Just reads as many characters as there are

b. So in particular, if we give this function enough characters and no new line, will just keep on writing over anything
c. If overwrite the return address, then it will jump to whatever address we put in that value
d. So we just subtract ret address value from other value
e. So can put address of bomb function, which is 0x8048502
f. So let's try to do that
g. Now the bomb function executes