Memory

- stack $\Longleftrightarrow$ heap, data, code
- command injection -
  `set args \perl -e 'print "A" x 24 . "\x4e\x06\x40\x00"'`
- can use gdb break points to get hex addressing
- payload generation - take syscall for exevp, call it with /bin/bash, assemble and inject binary into victim program

Assembly

- call pushes the address of next instruction (i.e., the return address) onto stack & transfers control to operand address
- ret jr, so jumps to return address
- leave sets stack pointer (%rsp) to frame pointer (%rbp) & sets frame pointer to saved frame pointer (popped from the stack)
- ret pops return address off stack & jumps to it
- rsp stack ptr, rbp- frame ptr
- we use AT&T syntax - source on left, destination on the right
- Result Flags: CF carry flag, PF parity flag, ZF zero flag, SF sign flag, OF overflow flag

General Registers:

Register	Description
%rax	Result register, also used in imul & idiv
%rbx	Miscellaneous register
%rcx	Fourth argument register
%rdx	Third argument register (used in imul & idiv)
%rsp	Stack pointer
%rbp	Frame pointer (base pointer)
%rsi	Second argument register
%rdi	First argument register
%r8	Fifth argument register
%r9	Sixth argument register
%r10	Miscellaneous register
%r11	Miscellaneous register
%r12-%r15	Miscellaneous register
%rip	Instruction pointer

Addressing Memory:

Syntax	Address	Description
(reg)	reg	Base addressing
d(reg)	reg + d	Base addressing + offset
d(reg,s)	(s x reg) + d	Scaled index + offset
d(reg1, reg2, s)	reg1 + (s x reg2) + 2	Base + scaled index + off.

Physical Security

- tamper resistance - key cannot be extracted. Expensive and not mass market
- tamper evidence - key extraction obvious, physically invasive attack only option
- self destruct baked into devices
- smart card - RFID external power
- probes into power analysis
- different operations take different power (DES cycles)
- stopping - can include dummy calculations, also isolate power source, add noise - extra operations, or random circuitry. Add delays, etc.

Security

- buffer overflow
- unbounded strcpy, buffer overflow, stack smashing
- protection magic canaries, no exec stack, using bounds checking for accesses
- NX - non executable stack
- How to get around non exec stack put stuff on heap. If NX heap, then disas memory, see where you are allowed to execute, and then jump to those instructions one by one. Basically assemble a program from bits of other programs.
- getting around a stack canary just read it, then include it in the overflow. OR read it from the location that serves as a golden canary. Problem need to make sure canaries match
- ASLR address space layout randomization hard, since then need different return address each time.
- tactics look for rsp movements, since that shows stack changes
- cal(q) important, since that is where rbp is written
- general tips for security analysis:
- notice off by one errors buff[10], but last index is 9 so write 1 byte past
- check inputs that attacker controls what if printing string with no null terminator, etc
- malloc/calloc make sure there is enough space, include space for null terminator
- strlen only checks for null terminator. So hex values will fully be parsed
- HW7 stuff - buffer overflow, insert shell code with nop slide and return addr into the middle of the slide
- shellcode is compiled binary `execvp(/bin/bash)` code

Side Channel Attacks

- timing - if password checks one char at a time, then can guess correct password since failure message comes quick
- power - voltage analyzer, carries take more power, etc
- Simple Power Analysis - use characteristics visible in a single trace - simple, exploitable operations have relationship with stuff seen in trace
- Differential Power Analysis - multiple traces, statistical methods - attacks intermediate results of algorithms
- stopping power analysis - make power consumption not dependent on process - noise, delays, extra power supply. randomize some values/operations. masking - algorithm modified - intermediate values not stored. operate on modified values, attempt to level power consumption to make it harder to distinguish
- sound - sound of keyboard typing, unique vibrations
- visual - reconstruct computer based on reflections
- wrench - human factor - coercion, bribing, etc

Q1: Consider the following attacks and counter-measures. a) *Spoofed web sites for phishing : X.509 certificates and TLS 1.2* Somewhat effective. If the user checks the certificate and validates the certificate chain, she can assure herself of the sites authenticity; however, few users actually do this.

b) *Session hijacking through stolen cookies acquired by eavesdropping : HTTPS* Very effective. An attacker cannot decrypt messages to steal cookies.

c) *Reflected XSS : same-origin policy* Ineffective. In this attack, the attackers code executes in the page of the good site, so the same-origin policy permits it.

d)*Stored XSS on a web-based bulletin board : escaping user-supplied data when generating HTML* Very effective. The malicious script will be displayed as script source, rather than being executed.

e) *SQL injection : escaping user-supplied data when generating HTML* Ineffective. SQL injection is enabled when server code fails to escape user supplied data when generating SQL to access a database.

f) *CSRF : HTTPS* Ineffective . This attack does not depend on the attacker eavesdropping, but rather depends on the browser automatically attaching cookies to requests

Q2 *In a sybil attack on a peer-to-peer network, a single controller runs a number of entities on the peer-to peer network. Give two (independent) examples of problems that a sybil attack on Bitcoin could create.*

- The attack can refuse to relay blocks and transactions from everyone, disconnecting you from the network
- The attacker can relay only blocks that he creates, making you vulnerable to double-spending attacks.
- The attacker can filter out certain transactions to execute a double-spending attack

Q3 For each security measures, explain might be attacked if DNS is compromised, or whether they are independent from DNS a) *HTTPS* X.509 certificates refer to a domain name, rather than to an IP address, and thus can be attacked if DNS is compromised. b) *Same-origin policy* Same-origin policy is based on domain names (not on IP addresses), and thus can be attacked if DNS is compromised. c) *Conventional packet-filter firewall policies* Firewall policies are written with respect to IP addresses, and thus are not dependent on DNS. **Q4** a) *In using RSA encryption, is it a problem if two users share the same composite modulus n?* Yes. They would both have the same factorization n=pq and both could find the others private key

b) *In using ElGamal encryption, is it a problem if two users share the same prime modulus?* No. The security only depends on the discrete log problem being hard modulo p. Even if two users share the modulus, they will not be able to find each others private key.

Q6 Consider a sybil attack on Tor to deanonymize users. Tor claims to have approximately 10,000 nodes in current use. Make the following simplifying assumptions: Tor choses a new random routing every 10 minutes (choosing a routing randomly chosen among all nodes with uniform distribution), Tor users use Tor 1000 hours a year, Tor users always use the same originating IP address, and Tor has three hops between the originating IP address and the destination IP address. Suppose that FAPSI (the Russian equivalent of the NSA) wants to deanonymize Tor users. Suppose that to support their effort, FAPSI has set up their sybil nodes, and they comprise 10% of all Tor nodes.

a) Over the course of the year, approximately what fraction of IP addresses using Tor will FAPSI observe. $6 * 1000 = 6000$ 10 minute slots of use per user each year, and during each 10 minute slot, an originating IP address has a 10% chance of being discovered. Over the course of a year, almost all originating IP addresses will be discovered.

b) Out of those IP addresses discovered in part a, over the course of a year, approximately what fraction of them will FAPSI be able to identify at least one outgoing IP address the incoming IP address is accessing. Again, $6 * 1000 = 6000$. To absolutely correlate an originating and outgoing IP address, FAPSI must control all the nodes in a 3-hop path. (That happens with probability $0.1 * 0.1 * 0.1 = .001$). Again, over the course of a year, almost all originating IP addresses will be correlated with at least one outgoing IP address.

Authors: Ivan Smirnov (http://ivansmirnov.name) and Tsion Behailu (http://tsion.me)

Now, let's work on understanding the assembly code. In fact, we'll assume that we have just the assembly code and no source code. The first thing to notice is that many opcodes end with a "q". That indicates that they are "quadword" (a word is 16 bits, so 64 bit) values. Some other suffixes to know are "b" (for byte), "w" (for word – 2 bytes), and "l" (for doubleword (long) – 4 byte values). So, all the values we are dealing with here 64 bit values – or what C (on 86-64 architectures) calls long. (On 32 bit architectures, a "long" was 32 bits – so that's why assembly code uses the "l" suffix for 32 bit values.)

Now, let's remember how registers are used. Notice that each register can be accessed as an eight-byte (quadword) value (e.g., **%rax**), a four-byte (doubleword) value (e.g., **%eax**), a two-byte (word) value (e.g., **%ax**) or a byte (e.g., **%al**) value. Notice that **%rbp** is reserved for use as the frame pointer and **%rsp** is reserved for use as the stack pointer.

```
pushq   %rbp
movq    %rsp, %rbp
```

This code begins every function. It saves the old frame pointer on the stack, and creates a new frame pointer by copying the stack pointer to the frame pointer register (**%rbp**).

```
bubblesort(long x, long y)
{
}
```

```
movq    %rdi, -40(%rbp)
movq    %rsi, -48(%rbp)
```

Now we are saving values on the stack. The stack grows down. We save the first argument at -40 from the frame pointer. We save the second argument at -48 from the frame pointer. This means that we have two long arguments (**x, y**) to the function. So far our reconstruction is

```
long a;
```

```
movq    $0, -8(%rbp)
```

```
a=0;
```

Now we are saving a value 0 to some variable on the stack. Thus we must have some long variable. So far our reconstruction is:

```
jmp     .L2
```

This jump instruction is typically used in some sort of control structure. Since there is not a test here, we can infer that it is a loop structure (rather than an if structure). Let's use the most general loop structure – the while statement. So far our reconstruction is:

```
while ()
{
}
```

```
.L5:  movq    -16(%rbp), %rax
      salq    $3, %rax
      addq    -40(%rbp), %rax
      movq    (%rax), %rdx
```

So now we are using the variable stored at **-16(%rbp)**, multiplying it by 8 (using a shift-arithmetic-left by 3 bits instruction), adding the address at **x** to it, and fetching the value at the resulting address into **%rdx**. This implies that **x** is an array, and since we multiplied it by 8, it must be an array of long values. So far our reconstruction is **/* calculate x[b] */**

```
addq    $1, %rax
salq    $3, %rax
addq    -40(%rbp), %rax
movq    (%rax), %rax
```

This is very similar except we are now computing **x[b+1]** and saving the result in **%rax**. So far our reconstruction is: **/* calculate x[b] and x[b+1] */**

```
if (x[b] > x[b+1])
{
}
```

```
cmpq    %rax, %rdx
jle     .L4
```

Now we compare **x[b]** and **x[b+1]**, and if **x[b] > x[b+1]**, we execute code (otherwise we skip ahead). This is a classic if statement. So far our reconstruction is:

```
movq    -16(%rbp), %rax
salq    $3, %rax
addq    -40(%rbp), %rax
movq    (%rax), %rax
movq    %rax, -24(%rbp)
```

As above, we find the value at **x[b]**, and now we store it in a new location (indicating a new variable), **-24(%rbp)**. So far our reconstruction is: **c = x[b];**

```
movq    -16(%rbp), %rax
salq    $3, %rax
addq    -40(%rbp), %rax
movq    -16(%rbp), %rdx
addq    $1, %rdx
salq    $3, %rdx
addq    -40(%rbp), %rdx
movq    (%rdx), %rdx
movq    %rdx, (%rax)
```

Using similar reasoning to above, we see that we are looking up the value at **x[b+1]** and put it in **x[b]**. So far our reconstruction is: **x[b] = x[b+1];**

```
.L4:
```

.L4:, indicating that this the end of the if statement. (If there were an else clause, we would expect a **jmp** opcode here. So far our reconstruction is: **}**

```
movq    -8(%rbp), %rax
movq    -48(%rbp), %rdx
movq    %rdx, %rcx
subq    %rax, %rcx
movq    %rcx, %rax
subq    $1, %rax
cmpq    -16(%rbp), %rax
jg      .L5
```

We move load **a** into **%rax** and **y** into **%rdx** (and then **%rcx**), and compute **y-a-1**, and see if it is greater than **b**. If so, we proceed with the while loop. Our reconstruction so far is: **while(b < y-a-1)**

```
popq    %rbp
ret
```

This is standard return code from a function. We pop the old values to the frame pointer, and return from the call.

One final question – does this code return a value? If so, it will be stored in %rax. This value is n-1. Now, we cannot tell from just this code whether the function returns n-1 or whether this just junk and the function returns nothing. Thus, we have two final possibilities:

```
void bubblesort(long x[], long y)
{
        long a, b, c;
        a=0;
        while (a < n-1)
        {
                b=0;
                while(b < y-a-1)
                {
                        if (x[b] > x[b+1])
                        {
                                c = x[b];
                                x[b] = x[b+1];
                                x[b+1] = c;
                        }
                        b++;
                }
                a++;
        }
}
```

```
bubblesort:
        pushq   %rbp
        movq    %rsp, %rbp
        movq    %rdi, -40(%rbp)
        movq    %rsi, -48(%rbp)
        movq    $0, -8(%rbp)
        jmp     .L2
.L6:
        movq    $0, -16(%rbp)
        jmp     .L3
.L5:
        movq    -16(%rbp), %rax
        salq    $3, %rax
        addq    -40(%rbp), %rax
        movq    (%rax), %rdx
        movq    -16(%rbp), %rax
        addq    $1, %rax
        salq    $3, %rax
        addq    -40(%rbp), %rax
        movq    (%rax), %rax
        cmpq    %rax, %rdx
        jle     .L4
        movq    -16(%rbp), %rax
        salq    $3, %rax
        addq    -40(%rbp), %rax
        movq    (%rax), %rax
        movq    %rax, -24(%rbp)
        movq    -16(%rbp), %rax
        salq    $3, %rax
        addq    -40(%rbp), %rax
        movq    -16(%rbp), %rdx
        addq    $1, %rdx
        salq    $3, %rdx
        addq    -40(%rbp), %rdx
        movq    (%rdx), %rdx
        movq    %rdx, (%rax)
.L4:
        addq    $1, -16(%rbp)
.L3:
        movq    -8(%rbp), %rax
        movq    -48(%rbp), %rdx
        movq    %rdx, %rcx
        subq    %rax, %rcx
        movq    %rcx, %rax
        subq    $1, %rax
        cmpq    -16(%rbp), %rax
        jg      .L5
        addq    $1, -8(%rbp)
.L2:
        movq    -48(%rbp), %rax
        subq    $1, %rax
        cmpq    -8(%rbp), %rax
        jg      .L6
        popq    %rbp
        ret
```

%r__ are 64 bit registers

%e__ are 32 bit registers

q suffix on instruction indicates a "quad-word" (64 bit) instruction

l suffix on instruction indicates a "long-word" (32 bit) instruction

%rbp is special register storing frame (base) ptr
%rsp is special register storing stack pointer
space between them is function's working space
stack grows down!

params stored -4 & -8 bytes below base pointer

C convention: always return value in **%eax** or **%rax**