Recap

- Application memory
  - Heap Memory
  - Call Stack
  - Globals
    - int w;
  - Instructions
    - push rbp
    - mov rbp, rsp

Heap Memory

Does not grow

Can grow while the application is running
Buffer Overflows

- When writing data to a buffer, you overrun into adjacent memory locations
- Often results in a crash, but sometimes can be exploited for other malicious behaviour, such as gaining elevated privileges
- Can occur on the stack:
  - Stack smashing
- Can occur on the heap:
  - Heap overflow
# Buffer Overflows

```c
#include <stdio.h>

int main ()
{
    int allow;
    char username[8];

    allow = 0;

    printf("Enter your username, please: ");
    gets(username);

    if (strcmp(username, "chris") == 0)
        allow = 1;

    if (allow)
        printf("Here is your private Bitcoin wallet: L2udm71vYECrgBcgZLA6JpUfUwDYHqcBA89Db9QazRYK Ghg1EbCZ\n");

    return 0;
}
```
Buffer Overflows

- Using a patched gcc which forces stack protection by default
- Deprecated APIs
- Helpful warnings

- jess can’t login
- chris can login
- .. but so can bbbbbbbbbbbbb
Buffer Overflows

```c
#include <stdio.h>

int main ()
{
    int allow;
    char username[8];

    allow = 0;

    printf("Enter your username, please: ");
    gets(username);

    if (strcmp(username, "chris") == 0)
        allow = 1;

    if (allow)
        printf("Here is your private Bitcoin wallet: ...\n");

    return 0;
}
```
Buffer Overflows

```c
#include <stdio.h>

int main ()
{
    int allow;
    char username[8];

    allow = 0;

    printf("Enter your username, please: ");
    gets(username);

    if (strcmp(username, "chris") == 0)
        allow = 1;

    if (allow)
        printf("Here is your private Bitcoin wallet: ...\n");

    return 0;
}
```
Buffer Overflows

```c
#include <stdio.h>

int main ()
{
    int allow;
    char username[8];

    allow = 0;

    printf("Enter your username, please: ");
    gets(username);

    if (strcmp(username, "chris") == 0)
        allow = 1;

    if (allow)
        printf("Here is your private Bitcoin wallet: ...");

    return 0;
}
```
Buffer Overflows

```c
#include <stdio.h>

int main ()
{
    int allow;
    char username[8];

    allow = 0;
    printf("Enter your username, please: ");
    gets(username);

    if (strcmp(username, "chris") == 0)
        allow = 1;

    if (allow)
        printf("Here is your private Bitcoin wallet: ...");

    return 0;
}
```
Buffer Overflows

.text (code segment) stores program instructions
.data segment stores global variables, static local variables
.rodata read-only data segment stores static constants

There are a few other segments try `objdump -s main.o`
.text (code segment) stores program instructions
.data segment stores global variables, static local variables
.rodata read-only data segment stores static constants

There are a few other segments try objdump -s main.o
Buffer Overflows

```c
#include <stdio.h>

int main ()
{
    int allow;
    char username[8];
    allow = 0;

    printf("Enter your username, please: ");
    gets(username);

    if (strcmp(username, "chris") == 0)
    {
        allow = 1;
    }

    if (allow)
    {
        printf("Here is your private Bitcoin wallet: ...");
    }

    return 0;
}
```
Buffer Overflows

```c
#include <stdio.h>

int main ()
{
    int allow;
    char username[8];
    allow = 0;
    printf("Enter your username, please: ");
    gets(username);
    if (strcmp(username, "chris") == 0)
        allow = 1;
    if (allow)
        printf("Here is your private Bitcoin wallet: ...");
    return 0;
}
```
Before calling \texttt{gets}, this is what the stack looks like:

```c
#include <stdio.h>

int main ()
{
    int allow;
    char username[8];

    allow = 0;

    printf("Enter your username, please: ");
    gets(username);

    if (strcmp(username, "chris") == 0)
        allow = 1;

    if (allow)
        printf("Here is your private Bitcoin wallet: ...);

    return 0;
}
```
Buffer Overflows

`gets` then fetches data into the address specified by `rax` without considering the bounds of the buffer it is putting it into...

```
#include <stdio.h>

int main ()
{
    int allow;
    char username[8];

    allow = 0;

    printf("Enter your username, please: ");
    gets(username);

    if (strcmp(username, "chris") == 0)
        allow = 1;

    if (allow)
        printf("Here is your private Bitcoin wallet: ...");

    return 0;
}
```
Buffer Overflows

For example: “jess”

```c
#include <stdio.h>

int main ()
{
    int allow;
    char username[8];

    allow = 0;
    printf("Enter your username, please: ");
    gets(username);

    if (strcmp(username, "chris") == 0)
        allow = 1;

    if (allow)
        printf("Here is your private Bitcoin wallet: ...");

    return 0;
}
```

The Stack

<table>
<thead>
<tr>
<th>ASCII</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
<td>106</td>
</tr>
<tr>
<td>e</td>
<td>101</td>
</tr>
<tr>
<td>s</td>
<td>115</td>
</tr>
<tr>
<td>s</td>
<td>115</td>
</tr>
<tr>
<td>0</td>
<td>-231</td>
</tr>
<tr>
<td>0</td>
<td>-532</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

main stack frame
Buffer Overflows

What about the string: “\text{abcd1234f}”

```
#include <stdio.h>

int main ()
{
    int allow;
    char username[8];

    allow = 0;
    printf("Enter your username, please: ");
    gets(username);

    if (strcmp(username, "chris") == 0)
        allow = 1;

    if (allow)
        printf("Here is your private Bitcoin wallet: ...");

    return 0;
}
```
Buffer Overflows

What about the string: “abcd1234f”

```
#include <stdio.h>

int main ()
{
    int allow;
    char username[8];
    allow = 0;

    printf("Enter your username, please: ");
    gets(username);

    if (strcmp(username, "chris") == 0)
        allow = 1;

    if (allow)
        printf("Here is your private Bitcoin wallet: ...");

    return 0;
}
```
Buffer Overflows

What about the string: “abcd1234f”

```c
#include <stdio.h>

int main ()
{
    int allow;
    char username[8];
    allow = 0;
    printf("Enter your username, please: ");
    gets(username);
    if (strcmp(username, "chris") == 0)
        allow = 1;
    if (allow)
        printf("Here is your private Bitcoin wallet: ...");
    return 0;
}
```

The Stack

<table>
<thead>
<tr>
<th>ASCII</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>97</td>
</tr>
<tr>
<td>b</td>
<td>98</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>int allow</td>
<td>-231</td>
</tr>
<tr>
<td>parent func rbp</td>
<td>-532</td>
</tr>
<tr>
<td>ret address</td>
<td></td>
</tr>
</tbody>
</table>
Buffer Overflows

What about the string: “abcd1234f”

```c
#include <stdio.h>

int main () {
    int allow;
    char username[8];
    allow = 0;
    printf("Enter your username, please: ");
    gets(username);
    if (strcmp(username, "chris") == 0)
        allow = 1;
    if (allow)
        printf("Here is your private Bitcoin wallet: ...\n");
    return 0;
}
```

<table>
<thead>
<tr>
<th>ASCII</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>97</td>
</tr>
<tr>
<td>b</td>
<td>98</td>
</tr>
<tr>
<td>c</td>
<td>99</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>0</td>
</tr>
<tr>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>g</td>
<td>0</td>
</tr>
<tr>
<td>h</td>
<td>0</td>
</tr>
<tr>
<td>i</td>
<td>-231</td>
</tr>
<tr>
<td>j</td>
<td>-532</td>
</tr>
</tbody>
</table>

The Stack

main stack frame
/include <stdio.h>

int main ()
{
    int allow;
    char username[8];

    allow = 0;
    printf("Enter your username, please: ");
    gets(username);

    if (strcmp(username, "chris") == 0)
        allow = 1;

    if (allow)
        printf("Here is your private Bitcoin wallet: ..."),
    return 0;
}
Buffer Overflows

What about the string: “abcd1234f”

```c
#include <stdio.h>

int main ()
{
    int allow;
    char username[8];

    allow = 0;

    printf("Enter your username, please: ");
    gets(username);

    if (strcmp(username, "chris") == 0)
        allow = 1;

    if (allow)
        printf("Here is your private Bitcoin wallet: ...");

    return 0;
}
```

---

The Stack

<table>
<thead>
<tr>
<th>ASCII</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>char username[0]</td>
<td>a 97</td>
</tr>
<tr>
<td>char username[1]</td>
<td>b 98</td>
</tr>
<tr>
<td>char username[2]</td>
<td>c 99</td>
</tr>
<tr>
<td>char username[3]</td>
<td>d 100</td>
</tr>
<tr>
<td>char username[4]</td>
<td>1 49</td>
</tr>
<tr>
<td>char username[5]</td>
<td>🦇 0</td>
</tr>
<tr>
<td>char username[6]</td>
<td>🦇 0</td>
</tr>
<tr>
<td>char username[7]</td>
<td>🦇 0</td>
</tr>
<tr>
<td>int allow</td>
<td>-231</td>
</tr>
<tr>
<td>parent func rbp</td>
<td>-532</td>
</tr>
<tr>
<td>ret address</td>
<td></td>
</tr>
</tbody>
</table>
Buffer Overflows

What about the string: “abcd1234f”

```c
#include <stdio.h>

int main ()
{
    int allow;
    char username[8];
    allow = 0;
    printf("Enter your username, please: ");
    gets(username);
    if (strcmp(username, "chris") == 0)
        allow = 1;
    if (allow)
        printf("Here is your private Bitcoin wallet: ...\n");
    return 0;
}
```

The Stack

<table>
<thead>
<tr>
<th>ASCII</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>char username[0]</td>
<td>a</td>
</tr>
<tr>
<td>char username[1]</td>
<td>b</td>
</tr>
<tr>
<td>char username[2]</td>
<td>c</td>
</tr>
<tr>
<td>char username[3]</td>
<td>d</td>
</tr>
<tr>
<td>char username[4]</td>
<td>1</td>
</tr>
<tr>
<td>char username[5]</td>
<td>2</td>
</tr>
<tr>
<td>char username[6]</td>
<td>0</td>
</tr>
<tr>
<td>char username[7]</td>
<td>0</td>
</tr>
<tr>
<td>int allow</td>
<td></td>
</tr>
<tr>
<td>parent func rbp</td>
<td>-231</td>
</tr>
<tr>
<td>ret address</td>
<td>-532</td>
</tr>
</tbody>
</table>
Buffer Overflows

What about the string: “abcd1234f"

```c
#include <stdio.h>

int main ()
{
    int allow;
    char username[8];
    allow = 0;
    printf("Enter your username, please: ");
    gets(username);
    if (strcmp(username, "chris") == 0)
        allow = 1;
    if (allow)
        printf("Here is your private Bitcoin wallet: ... ");
    return 0;
}
```
Buffer Overflows

What about the string: “abcd1234f”

```c
#include <stdio.h>

int main ()
{
    int allow;
    char username[8];
    allow = 0;

    printf("Enter your username, please: ");
    gets(username);

    if (strcmp(username, "chris") == 0)
        allow = 1;

    if (allow)
        printf("Here is your private Bitcoin wallet: ...");

    return 0;
}
```

The Stack

<table>
<thead>
<tr>
<th>ASCII</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>char username[0]</td>
<td>a</td>
</tr>
<tr>
<td>char username[1]</td>
<td>b</td>
</tr>
<tr>
<td>char username[2]</td>
<td>c</td>
</tr>
<tr>
<td>char username[3]</td>
<td>d</td>
</tr>
<tr>
<td>char username[4]</td>
<td>1</td>
</tr>
<tr>
<td>char username[5]</td>
<td>2</td>
</tr>
<tr>
<td>char username[6]</td>
<td>3</td>
</tr>
<tr>
<td>char username[7]</td>
<td>4</td>
</tr>
<tr>
<td>int allow</td>
<td>0</td>
</tr>
<tr>
<td>parent func rbp</td>
<td>-231</td>
</tr>
<tr>
<td>ret address</td>
<td>-532</td>
</tr>
</tbody>
</table>
What about the string: “abcd1234f”

```c
#include <stdio.h>

int main ()
{
    int allow;
    char username[8];

    allow = 0;
    printf("Enter your username, please: ");
    gets(username);
    if (strcmp(username, "chris") == 0)
        allow = 1;
    if (allow)
        printf("Here is your private Bitcoin wallet: ...\n");
    return 0;
}
```
Buffer Overflows

What about the string: “**abcd1234f**”

We can write over the `allow` variable data

```c
#include <stdio.h>

int main ()
{
    int allow;
    char username[8];

    allow = 0;
    printf("Enter your username, please: ");
    gets(username);
    if (strcmp(username, "chris") == 0)
        allow = 1;
    if (allow)
        printf("Here is your private Bitcoin wallet: ...");
    return 0;
}
```

<table>
<thead>
<tr>
<th>ASCII</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>97</td>
</tr>
<tr>
<td>b</td>
<td>98</td>
</tr>
<tr>
<td>c</td>
<td>99</td>
</tr>
<tr>
<td>d</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>49</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
</tr>
<tr>
<td>3</td>
<td>51</td>
</tr>
<tr>
<td>4</td>
<td>52</td>
</tr>
<tr>
<td>f</td>
<td>102</td>
</tr>
<tr>
<td>-231</td>
<td>-532</td>
</tr>
</tbody>
</table>

The Stack

- `main` stack frame
- `parent func rbp`
- `ret address`

ASCII

Decimal

We can write over the `allow` variable data.
Buffer Overflows

At this point, allow is now: **102**

```
#include <stdio.h>

int main ()
{
    int allow;
    char username[8];

    allow = 0;
    printf("Enter your username, please: ");
    gets(username);
    if (strcmp(username, "chris") == 0)
        allow = 1;
    if (allow)
        printf("Here is your private Bitcoin wallet: ...");
    return 0;
}
```
Buffer Overflows

String check fails, allow is still: **102**

```c
#include <stdio.h>

int main ()
{
    int allow;
    char username[8];
    allow = 0;
    printf("Enter your username, please: ");
    gets(username);
    if (strcmp(username, "chris") == 0)
        allow = 1;
    if (allow)
        printf("Here is your private Bitcoin wallet: ...");
    return 0;
}
```

<table>
<thead>
<tr>
<th>ASCII</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>97</td>
</tr>
<tr>
<td>b</td>
<td>98</td>
</tr>
<tr>
<td>c</td>
<td>99</td>
</tr>
<tr>
<td>d</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>49</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
</tr>
<tr>
<td>3</td>
<td>51</td>
</tr>
<tr>
<td>4</td>
<td>52</td>
</tr>
<tr>
<td>f</td>
<td>102</td>
</tr>
<tr>
<td>-231</td>
<td>-532</td>
</tr>
</tbody>
</table>

The Stack

- `username[0]`: `f`
- `username[1]`: `-231`
Buffer Overflows

#include <stdio.h>

int main ()
{
    int allow;
    char username[8];
    allow = 0;
    printf("Enter your username, please: ");
    gets(username);
    if (strcmp(username, "chris") == 0)
        allow = 1;
    if (allow)
        printf("Here is your private Bitcoin wallet: ...");
    return 0;
}
Protection against Buffer Overflows

- Detect and abort before malicious behaviour occurs:

```
chris@chris-lab ~/security $ master • gcc -fstack-protector -std=gnu11 -O0 -pedantic main.c -o main.o

main.c: In function `main':
main.c:11:5: warning: `gets' is deprecated [-Wdeprecated-declarations]
    gets(username);
    ^~~~
In file included from main.c:1:0:
/usr/include/stdio.h:577:14: note: declared here
extern char *gets (char *__s) __wur __attribute_deprecated__;
    ^~~~
/tmp/ccyFWBuE.o: In function `main':
main.c:(.text+0x48): warning: the `gets' function is dangerous and should not be used.
chris@chris-lab ~/security $ master • ./main.o
Enter your username, please: abcd1234f
*** stack smashing detected ***: <unknown> terminated
4063 abort (core dumped) ./main.o
x chris@chris-lab ~/security $ master •
```
Protection against Buffer Overflows

```c
#include <stdio.h>
#include <stdlib.h>

int main ()
{
    int allow;
    char* username;
    allow = 0;
    username = malloc(8 * sizeof(*username));
    if (!username)
        return 1;
    printf("Enter your username, please: ");
    fgets(username, 8, stdin);
    strtok(username, "\n");
    if (strcmp(username, "chris") == 0)
        allow = 1;
    free(username);
    if (allow)
        printf("Here is your private Bitcoin wallet: L2udm71vYECrgBcgZLA6JpUfUwDYHqcBA89Db9QazRYKghg1EbCZ\n");
    return 0;
}
```

Use heap memory

Do proper bounds checking (no more than 8 characters)
Protection against Buffer Overflows

- Using heap memory and `fgets`
Other common functions vulnerable to overflow (and their mitigations) not examined

Some more potentially dangerous system calls...

- **gets** - read line from stdin
- **strcpy** - copies string src dst
- **strcat** - appends string src dst
- **sprintf** - write data to string buffer
- **scanf** - read data from stdin
- **sscanf** - read data from string
- **fscanf** - read data from stream
- **vfscanf** - read from stream to args
- **realpath** - returns absolute path
- **getenv** - get environment string
- **getpass** - gets a password

... and lots more in many languages...
Heartbleed: A Buffer Over-read

Buffer over-read vulnerability in OpenSSL

- Open source code that handles a large proportion of the world's secured web traffic
  - Traffic between you and banks
  - Private emails
  - Social networks
- Clients send heartbeats to servers (are you alive?)
- Server responds with data
- A particular version of OpenSSL didn't check for over-read
- Each heartbeat could reveal 64k of application memory
  - Lots of sensitive data leaked
  - Big websites request password resets following heartbleed
    - Reddit, Github, Bitbucket, Mojang, Amazon AWS, Pinterest, Tumblr, ...
Stack Smashing

- So we can overwrite memory. What else can we do?

<table>
<thead>
<tr>
<th>The Stack</th>
<th>ASCII</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>char username[0]</td>
<td>a</td>
<td>97</td>
</tr>
<tr>
<td>char username[1]</td>
<td>b</td>
<td>98</td>
</tr>
<tr>
<td>char username[2]</td>
<td>c</td>
<td>99</td>
</tr>
<tr>
<td>char username[3]</td>
<td>d</td>
<td>100</td>
</tr>
<tr>
<td>char username[4]</td>
<td>1</td>
<td>49</td>
</tr>
<tr>
<td>char username[5]</td>
<td>2</td>
<td>50</td>
</tr>
<tr>
<td>char username[6]</td>
<td>3</td>
<td>51</td>
</tr>
<tr>
<td>char username[7]</td>
<td>4</td>
<td>52</td>
</tr>
<tr>
<td>int allow</td>
<td>f</td>
<td>102</td>
</tr>
<tr>
<td>parent func rbp</td>
<td></td>
<td>-231</td>
</tr>
<tr>
<td>ret address</td>
<td></td>
<td>-532</td>
</tr>
</tbody>
</table>

What if we kept going until we reached here?
Choose where we want to jump to...
Stack Smashing

● What if we jumped to somewhere else where we had malicious code?
  ○ If we can use this on a program that has higher privilege than ourself, we can jump to deployed shellcode for that level of privilege.
  ○ Shell code is executable code inserted as a payload for insertion attacks.

● Countermeasures:
  ○ Check buffer lengths
  ○ Use heap memory
  ○ Use ASLR, on the fly randomization of memory to make buffer flow attacks more difficult to implement.
    ■ Similar concept of making the operating system less predictable and much harder to do these kinds of attacks.
    ■ Has been bypassed using side-channel attacks (2017)
  ○ Use a canary

● We can do similar things on the heap
## Canary Value

The Stack

<table>
<thead>
<tr>
<th>char username[0]</th>
<th>a</th>
<th>97</th>
</tr>
</thead>
<tbody>
<tr>
<td>char username[1]</td>
<td>b</td>
<td>98</td>
</tr>
<tr>
<td>char username[2]</td>
<td>c</td>
<td>99</td>
</tr>
<tr>
<td>char username[3]</td>
<td>d</td>
<td>100</td>
</tr>
<tr>
<td>char username[4]</td>
<td>1</td>
<td>49</td>
</tr>
<tr>
<td>char username[5]</td>
<td>2</td>
<td>50</td>
</tr>
<tr>
<td>char username[6]</td>
<td>3</td>
<td>51</td>
</tr>
<tr>
<td>char username[7]</td>
<td>4</td>
<td>52</td>
</tr>
</tbody>
</table>

### main stack frame

| int canary | 9315 |
| parent func rbp | -231 |
| ret address | -532 |

Generate random number just before stack return pointer.
### Canary Value

After writing to buffer, check if value is same as randomly generated value:

$$102 \neq 9315$$

...this has been tampered with, exit!
Heap Smashing and NOP slides

- Heap memory rarely contains pointers that influence control flow
  - Needs to be combined as part of larger attack
  - Has been used in practice in some popular software
    - Internet Explorer
    - VLC multimedia player
    - Adobe Acrobat
    - Adobe Flash

- Heap sprays
  - Attempts to put a certain sequence of bytes at a predetermined location in the memory by allocating large blocks on the process’s heap and filling the bytes in these blocks with specific values.

- NOP slides (NOP sleds)
  - “Move onto next instruction” - put loads of x90’s followed by your shell code. Then your return address is likely to hit one and slide to the malicious code.
Race Condition Attacks

- Occur when multiple processes or threads operate on shared data.
- Attacks occur in many different situations:
  - Typically developers perform two or more steps but forget that hackers can **do something malicious in the gap** between the steps
- Popular example, **Dirty Cow**

Exploits copy-on-write (CoW) functionality in OS to gain root (**quite easy to do**).

- Two processes may read same physical memory.
- If one tries to write, the OS makes a copy.
- Dirty cow map sensitive files that you want to modify, invokes CoW, opens two threads which interfere and allow you to write over sensitive file.
Timing Attacks

```cpp
bool check_password(string real, string guess) {
    for (int i=0; i<16; ++i)
        if (real[i] != guess[i])
            return false;
    return true;
}
```

You would typically need $96^{16}$ guesses to brute force

$$= 52,040,292,466,647,269,602,037,015,248,896$$

However if you accurately time application, it finishes at different times

Timing attack:

1. Try each of 96 chars for first letter in a random 16-length string.
   ○ Find which character takes longest to return false.
2. Move on to next character, and repeat

Would only take $96 \times 16$ guesses = maximum of $1,536$ attempts to brute force
Impact of AI on Cyber Security

- Gaining recent research traction (especially in 2018)
- **Lots of unethical research** taking place in this field
  - Shown to accurately predict sensitive information about people
  - Why are you researching that? “It doesn't matter, it's not personally identifiable.” Yes it does!
  - “When a model is trained, you can't reverse engineer it to recover source data, so its ok!”
- Will it advance faster than we can keep up with?

More/bigger datasets becoming available:

https://www.secrepo.com/

Questions:

- Will AI **help us write more secure software**?
- ...or will AI discover more vulnerabilities faster than we can patch?
Impact of AI on Cyber Security

- Better anti-virus or...
- Better virus?

Good at analysis of complex ill-defined problems - large systems/datasets
Impact of AI on Cyber Security

Adversarial against Humans:

- Chatbots
  - New types of spear phishing?
- Can't distinguish human from AI voice
- Reinforcement learning (Alpha Go, Starcraft)?
  - Real-world game?
- Deep Learning fools CAPTCHA

Fraudster Chatbots

Currently mostly focusing on detecting malware and intrusions

Collection of AI Cyber Security Research Papers
Better Defense Example:
Malware Prediction

Kaggle competition launched 18 hours ago:

https://www.kaggle.com/c/microsoft-malware-prediction

- ProductName - Defender state information e.g. win8defender
- EngineVersion - Defender state information e.g. 1.1.12603.0
- AppVersion - Defender state information e.g. 4.9.10586.0
- AvSigVersion - Defender state information e.g. 1.217.1014.0
- IsBeta - Defender state information e.g. false
- AVProductsInstalled - NA
- AVProductsEnabled - NA
- CountryIdentifier - ID for the country the machine is located in
- CityIdentifier - ID for the city the machine is located in
- OrganizationIdentifier - ID for the organization the machine belongs in, organization ID is mapped to both specific companies and broad industries
- GeoNameldentifier - ID for the geographic region a machine is located in
Fig. 1: Summary of PassGAN’s Architecture. In the training procedure, shown in (a), the discriminator (D) processes passwords from the training dataset, as well as password samples produced by the generator (G). Based on the feedback from D, G fine-tunes its network to produce password samples that are close to the training set (G has no direct access to the training set). The password generation procedure is shown in (b).

Recall from Lecture 1

An arms race between humans...

- Bad user
- Brute force
- Passwords leaked
- Rainbow tables
- DoS...

Unsecure System
- 2014: 3 users
- 2015: 150 users
- 2016: 3k users
- 2017: 10k users
- 2018: 100k users

An arms race between data/compute?

image from https://mile2.com
Questions?

- What to revise
- Research themes
- Any concepts discussed which were challenging to understand?
Romantic Takeaways

- **Just remember how easy it is to break in with a little time and effort**
  - Don’t assume people won’t try to break your code
- **Security covers all levels and infrastructure of a system**
  - The weakest link
  - Check your permissions
  - Use strong passwords
  - Sanitize your inputs
  - Prepare your statements
  - Lock your machine physically
- **Understand the platform**
- **Understand the people**
- **Don't be careless or manage in a way that promotes carelessness**
- **KISS!**
Just remember how easy it is to break in with a little time and effort
  ○ Don’t assume people won’t try to break your code

Security covers all levels and infrastructure of a system
  ○ The weakest link
  ○ Check your permissions
  ○ Use strong passwords
  ○ Sanitize your inputs
  ○ Prepare your statements
  ○ Lock your machine physically

Understand the platform

Understand the people

Don’t be careless or manage in a way that promotes carelessness

KISS!