Computer Security and Privacy

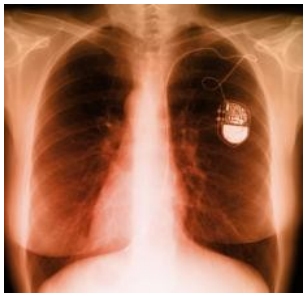
AIMS Senegal
Winter 2016
Instructor: Tara Whalen

Topic #2:
Software Security

Security: Not Just for PCs

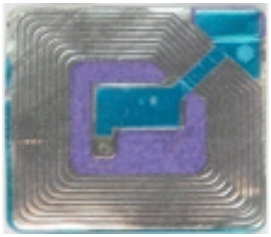
smartphones

voting machines



medical devices

wearables



RFID

cars

game platforms

airplanes

Software Lifecycle (Simplified)

- Requirements
- Design
- Implementation
- Testing
- Use

Software Problems are Ubiquitous

Software Bug Halts F-22 Flight

Posted by kdawson on Sunday February 25, @06:35PM
from the **dare-you-to-cross-this-line** dept.

mgh02114 writes

> "The new US stealth fighter, the F-22 Raptor, was deployed for the first time to Asia earlier this month. On Feb. 11, twelve Raptors flying from Hawaii to Japan were forced to turn back when a software glitch crashed all of the F-22s' on-board computers as they crossed the international date line. The delay in arrival in Japan was previously reported, with rumors of problems with the software. CNN television, however, this morning reported that every fighter completely lost all navigation and communications when they crossed the international date line. They reportedly had to turn around and follow their tankers by visual contact back to Hawaii. According to the CNN story, if they had not been with their tankers, or the weather had been bad, this would have been serious. CNN has not put up anything on their website yet."

(2007)

Software Problems are Ubiquitous

1985-1987 -- Therac-25 medical accelerator. A radiation therapy device malfunctions and delivers lethal radiation doses at several medical facilities. Based upon a previous design, the Therac-25 was an "improved" therapy system that could deliver two different kinds of radiation: either a low-power electron beam (beta particles) or X-rays. The Therac-25's X-rays were generated by smashing high-power electrons into a metal target positioned between the electron gun and the patient. A second "improvement" was the replacement of the older Therac-20's electromechanical safety interlocks with software control, a decision made because software was perceived to be more reliable.

What engineers didn't know was that both the 20 and the 25 were built upon an operating system that had been kludged together by a programmer with no formal training. Because of a subtle bug called a "race condition," a quick-fingered typist could accidentally configure the Therac-25 so the electron beam would fire in high-power mode but with the metal X-ray target out of position. At least five patients die; others are seriously injured.

Software Problems are Ubiquitous

January 15, 1990 -- AT&T Network Outage. A bug in a new release of the software that controls AT&T's #4ESS long distance switches causes these mammoth computers to crash when they receive a specific message from one of their neighboring machines -- a message that the neighbors send out when they recover from a crash.

One day a switch in New York crashes and reboots, causing its neighboring switches to crash, then their neighbors' neighbors, and so on. Soon, 114 switches are crashing and rebooting every six seconds, leaving an estimated 60 thousand people without long distance service for nine hours. The fix: engineers load the previous software release.

Adversarial Failures

- Software bugs are bad
  - Consequences can be serious
- Even worse when an intelligent adversary wishes to exploit them!
  - Intelligent adversaries: Force bugs into "worst possible" conditions/states
  - Intelligent adversaries: Pick their targets

When is a program secure?

- What does that even mean?

When is a program secure?

- When it does exactly what it should?
  - Not more.
  - Not less.
- But how do we know what a program is supposed to do?
  - Somebody tells us? (But do we trust them?)
  - We write the code ourselves? (But what fraction of the software you use have you written?)

When is a program secure?

- Let's try again.
- "A program is secure when it doesn't do bad things"
- Well, what are "bad things?"
- Could specify a list of "bad" things to avoid:
  - Delete or corrupt important files
  - Crash my system
  - Send threatening email to the course professor
- But… what if most of the time the program doesn't do bad things, but occasionally it does? Or *could*? Is it secure?

When is a program secure?

- Remember: Perfect security does not exist
- Security vulnerabilities (in software) are the result of violating an assumption about the software
  - (or, more generally the entire system)
- Corollary: As long as you make assumptions, you're vulnerable.
- And: You always need to make assumptions! (or else your software is useless and/or slow)
- **However, some assumptions are more dangerous than others!**

What is a software vulnerability?

- One definition: A bug in a software program that allows an unprivileged user capabilities that should be denied to them
- Most problematic kind?
  - "Control flow hijacking"
  - Bug that allows input data to be executed as code
- Example: Buffer overflows

Vulnerability: Buffer overflows

Buffer overflow bugs:

- ***Big*** class of bugs
  - Normal conditions: Can sometimes cause systems to fail
  - Adversarial conditions: Attacker able to violate security of a system (control it, obtain private information, …)

A Bit of History: Morris Worm

- Worm was released in 1988 by Robert Morris
  - Graduate student at Cornell, son of NSA chief scientist
  - Convicted under Computer Fraud and Abuse Act, sentenced to 3 years of probation and 400 hours of community service
  - Now an EECS professor at MIT
- Worm was intended to propagate slowly and harmlessly measure the size of the Internet
- Due to a coding error, it created new copies as fast as it could and overloaded infected machines
- $10-100M worth of damage

Morris Worm and Buffer Overflow

- One of the worm's propagation techniques was a buffer overflow attack against a vulnerable version of software ("`fingerd`") on VAX systems
  - By sending special string to `fingerd`, worm caused it to execute code creating a new worm copy
  - Unable to determine what type of operating system was running on remote systems (VAX?), worm also attacked `fingerd` on Suns running BSD, causing them to crash (instead of spawning a new copy)

… And More

- Conficker (2008-08): overflow in Windows RPC
  - Around 10 million machines infected (estimates vary)
- Stuxnet (2009-10): several zero-day overflows + same Windows RPC overflow as Conficker
  - Windows print spooler service
  - Windows LNK shortcut display
  - Windows task scheduler
- Flame (2010-12): same print spooler and LNK overflows as Stuxnet
  - Targeted cyperespionage virus
- Still ubiquitous, especially in embedded systems

Attacks on Memory Buffers

- Buffer is a pre-defined data storage area inside computer memory (stack or heap)
- Typical situation:
  - A function takes some input that it writes into a pre-allocated buffer.
  - The developer forgets to check that the size of the input isn't larger than the size of the buffer.
  - Uh oh.
    - "Normal" bad input: crash
    - "Adversarial" bad input : take control of execution

Overflow example

- Suppose Web server contains this function

```
void func(char *str) {
        char buf[126];
        ...
        strcpy(buf,str);
        ...
    }
```

"string copy (destination, source)"

- No bounds checking on strcpy()
- If str is longer than 126 bytes?
  - Program may crash
  - Attacker may change program behavior

Main cause: memory management problems

- Happens when program is responsible for its memory management (C/C++)
- Memory management is very error-prone
  - *Very common error in C/C++: program crashes with a "segmentation fault"*
- Technical term: C and C++ do not offer **memory-safety**

Risks of untrusted input

Any C(++) code acting on untrusted input is at risk

- code taking input over untrusted network
  - e.g., sendmail, web browser, wireless network driver,...
- code acting on untrusted files (downloaded, emailed)
- also embedded software
  - e.g., in devices with (wireless) network connections such as mobile phones, RFID card, airplane navigation systems…

Main issue: memory management in C/C++

- Typical bugs:
  - Writing past the bound of an array
  - Many problems with memory management
    - e.g.,: missing initialization
- For efficiency, these bugs are not detected at run time:
  - behaviour of a buggy program is *undefined*

Stack Overflows

How do you get control of a system through these bugs?

1. Get your code into victim's address space (memory) - *placement*

2. Get their program to jump there - *diversion*

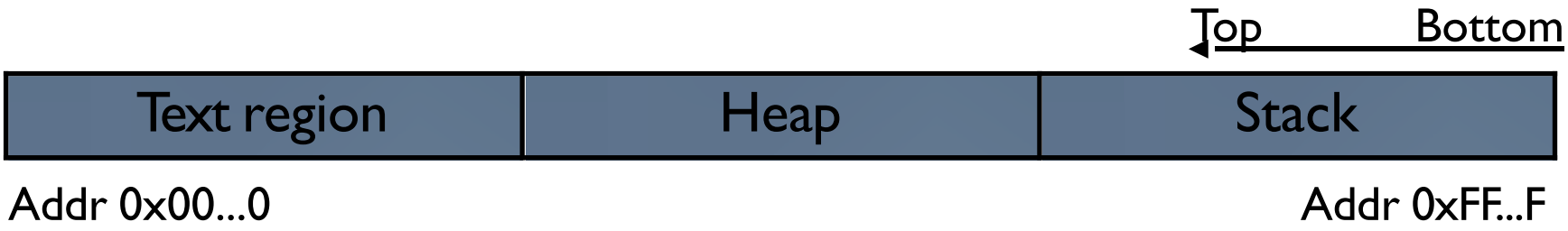
- In this course we will look at **stack overflows** as a type of buffer overflow

Stack Overflows

- In order to understand this attack, you need to understand a *little* about how functions are handled in memory (the "call stack")
- It's more important to get the basic idea than to worry about all the details

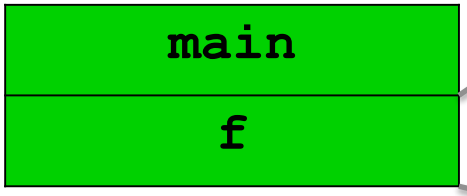
Memory Layout

- Text region: Executable code of the program
- Heap: Dynamically allocated data
- Stack: Local variables, function return addresses; grows and shrinks as functions are called and return

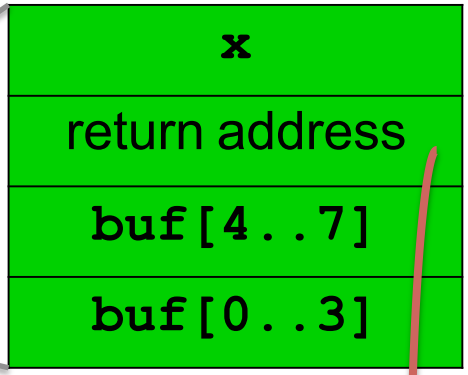


Call Stack: a (very) simple example

The stack consists of **frames**: one per function



Contents of a frame:



function arguments

where to go to continue program when function ends

local variables

```
void f(int x) {
char[8]buf;
…
}

void main() {
   f(…);
   ...
}
...
```

Call Stack: a (very) simple example

When function **f** is completed, frame is "popped off" the stack (cleared) – we don't need it anymore

main

```
void f(int x) {
char[8] buf;
…
}

void main() {
f(…);
…
}
…
```

When we next encounter another function in the program, it will get "pushed" on to the stack…

Stack Buffers

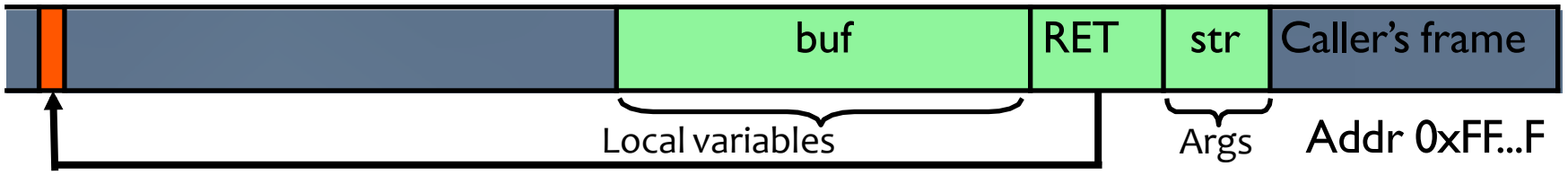
- Suppose Web server contains this function:

```
void func(char *str) {
        char buf[126];
        strcpy(buf,str);
    }
```

Allocate local buffer
(126 bytes reserved on stack)

Copy argument into local buffer

- When this function is invoked, a new frame is pushed onto the stack.



Execute code at this address after func() finishes

NB: diagram is simplified – actual call stacks have a little bit more complexity

What if Buffer is Overstuffed?

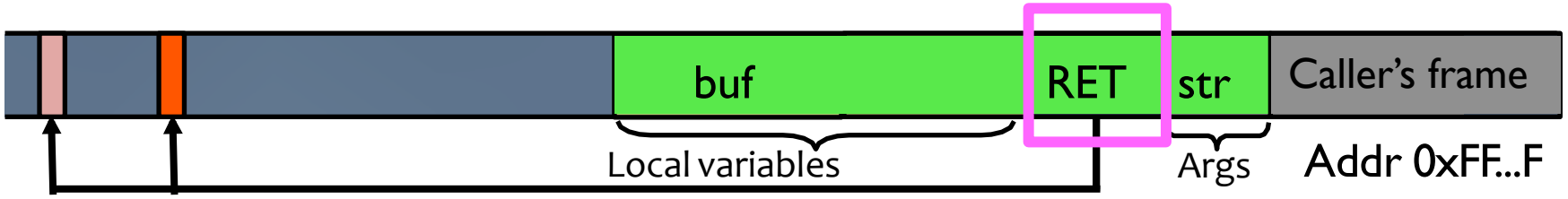
- Memory pointed to by str is copied onto stack…

```
void func(char *str) {
        char buf[126];
        strcpy(buf,str);
    }
```

strcpy does NOT check whether the string at *str contains fewer than 126 characters

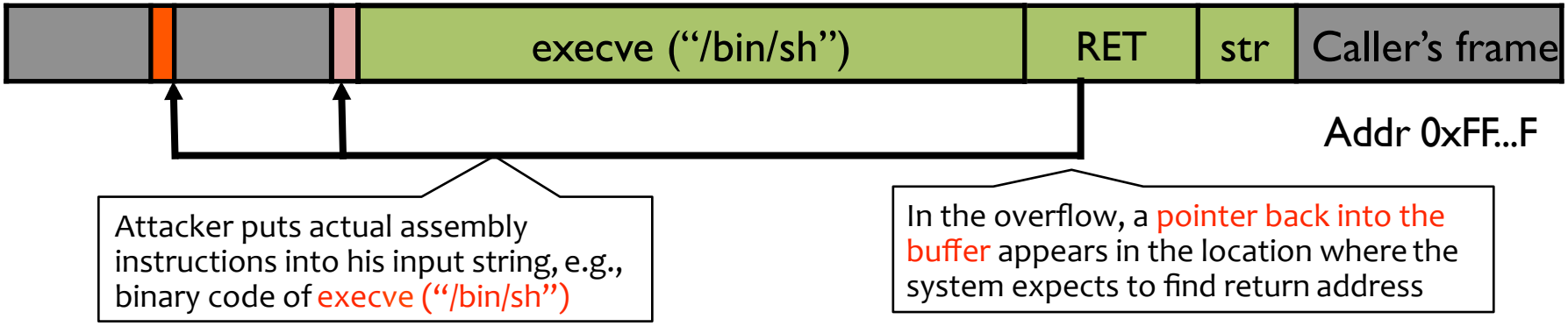
- If a string longer than 126 bytes is copied into buffer, it will overwrite adjacent stack locations.

This will be interpreted as return address!



Executing Attack Code

- Suppose buffer contains attacker-created string
  - For example, str points to a string received from the network as the URL



- When function exits, code in the buffer will be executed, giving attacker a shell
  - Root shell if the victim program is setuid root

Recurring problem: mixing control & data

Related example:

- In 1950s, Joe Engressia showed the telephone network could be hacked by phone phreaking:
    - i.e., whistling at particular frequencies
- The root cause: in-band signaling
    - Tone dialing: can send control signals to phone
        - free long-distance calls!
        - (no, this does not work anymore…)

Buffer Overflow Issues

- Executable attack code is stored on stack, inside the buffer containing attacker's string
  - Stack memory is supposed to contain only data, but…
- Overflow portion of the buffer must contain correct address of attack code in the RET position
- The value in the RET position must point to the beginning of attack assembly code in the buffer
  - Otherwise application will (probably) crash with segmentation violation
- Attacker must correctly guess in which stack position their buffer will be when the function is called
- Not trivial to do: depends on specifics of the system

Problem: No Bounds Checking

- strcpy does ***not*** check input size
  - strcpy(buf, str) simply copies memory contents into buf until "\0" (null) is encountered, ignoring the size of area allocated to buf
- Many C library functions are unsafe
  - strcpy (char *dest, const char *src)
  - strcat (char *dest, const char *src)
  - gets (char *s)
  - scanf (const char *format, …)
  - printf (const char *format, …)

Example: gets

```
char buf[20];
gets(buf); // read user input until
           // first EoL or EoF character
```

- *Never* use **gets**
- Use **fgets (buf, size, stdin)** instead

Example: strcpy

```
char dest[20];
strcpy(dest, src); // copies string src to dest
```

- **strcpy** assumes **dest** is long enough, and assumes **src** is null-terminated (\0 at the end)
- Use **strncpy (dest, src, size)** instead

Find the bug!

```
char buf[20];
char prefix[] = "http://";
char path[] = "www.example.com";
...
strcpy(buf, prefix);
// copies the string prefix to buf
strncat(buf, path, sizeof(buf));
// concatenates path to the string buf (adds to end)
// syntax: strncat (dest, src, size)
```

Find the bug!

```
char buf[20];
char prefix[] = "http://";
char path[] = "www.example.com";
...
strcpy(buf, prefix);
// copies the string prefix to buf
strncat(buf, path, sizeof(buf));
// concatenates path to the string buf (adds to end)
// syntax: strncat (dest, src, size)
```

strncat's 3rd parameter is number of chars to copy, not the buffer size

Another common mistake is giving **sizeof(path)** as 3rd argument...

Defenses against buffer overflows

Two basic approaches:

- Try to eliminate bugs in the code before it runs
- Try to keep bugs from causing the memory problem, if they occur during execution

Usually use combination of different approaches

Defenses against buffer overflows

Approach #1: Try to eliminate bugs in the code before it runs

Solution: be a perfect programmer? (Very unlikely!)

Instead:

"Compile time" approaches

- Memory-safe languages
- Testing

(Absence of) Language-Level Security

Buffer overflows are something we see in C/C++

Note that in other programming languages, we might not have to worry about:

- writing past array bounds (the language will notify you of the error)
- lots of memory management issues (e.g., takes care of initializing memory for you)
- ...

Tony Hoare on Design Principles of ALGOL 60

In his Turing Award lecture

> "The first principle was *security: ...* A consequence of this principle is that *every subscript was checked at run time against both the upper and the lower declared bounds of the array*. Many years later we asked our customers whether they wished us to provide an option to switch off these checks in the interests of efficiency. Unanimously, they urged us not to - they knew how frequently subscript errors occur on production runs where failure to detect them could be disastrous.
>
> *I note with fear and horror that even in 1980, language designers and users have not learned this lesson. In any respectable branch of engineering, failure to observe such elementary precautions would have long been against the law*."

[C.A.R. Hoare, The Emperor's Old Clothes, Communications of the ACM,1980]

Choose another language

- Write programs in memory-safe languages
  - Java, C#, Haskell, etc.
- Type system prevents buffer overflow from being expressed
- Limitations:
  - Not always an option
  - Legacy code issues, performance, other requirements may mean you must use C/C++

Code Testing

Two common approaches:

- "Fuzzing"
- Static analysis

Code Testing: Fuzzing

(also called "Fuzz Testing")

- Deals with the problem of untrusted input
- Idea: test a lot of kinds of input!
- Generate "random" inputs to program
  - Not entirely random, but tries a lot of combinations
- See if program crashes
- If it crashes: found a bug!
- Bug may be exploitable

Code Testing: Fuzzing

- Surprisingly effective
- Now standard part of development lifecycle
- Limitations:
  - Usually only finds simple bugs
  - Not finding a bug does not mean there is no bug

Code Testing: Static Analysis

Static analysis: automated code testing

Two types: easy and complex

Easy: "the type you write in 100 lines of python"

- look for known patterns and issues
- example: unsafe string functions: strcpy(), sprintf(), gets()

Complex: "the type you get a PhD for"

- buy this (From Coverity, Fortify, etc.)
- built into Visual Studio
- expert programmers can make their own

Code Testing: Static Analysis

- Limitations of static analysis:
  - High false positive rates
  - Many properties cannot be easily modeled
  - Difficult to build
  - Almost never have all source code in real systems (operating system, shared libraries, etc.)
- If you want to experiment
  - Free tools available: PreFast, RATS…
  - Can try with open source code

Defenses against buffer overflows

- Approach #2: Try to keep bugs from causing the memory problem, if they occur during execution

"Run time" approaches

- Prevent data from taking control
  - Stack validation (stack canaries)
  - Memory protection
- Protect jump target (can't find it)
  - Randomization

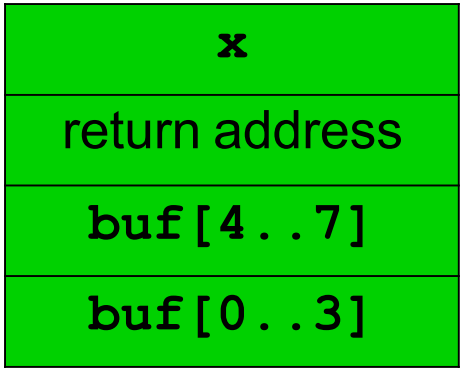
Stack Canaries

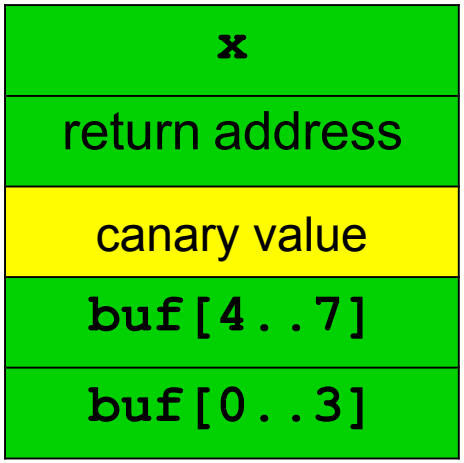
- (sometimes called "stack cookie" – I usually see "canary")
- introduced in StackGuard in compiler (gcc)
- A "sentinel value" – the stack canary – is written on the stack in front of the return address and checked when function returns
- a careless stack overflow will overwrite the canary, which can then be detected
  - verify the value prior to function return

Aside: Why “canary”?

Stack Canaries

- example of canary on simple stack





Stack Canaries

- Limits of stack canary:
- a careful attacker can overwrite the canary with the correct value
- additional countermeasures:
  - use a random value for the canary
  - Initialized at program startup
  - different each time program is run
  - assumption: adversary doesn't know it

Stack Canaries

- Limitations of stack canary:
  - Does not cover all types of memory attacks

Memory protection

- Recall our problem:
  - Having data + control (instructions) on the stack
  - What if we could prevent code from running from the stack?
- Solution: non-executable memory
- Distinguish
  - X: executable memory (for storing code)
  - W: writeable, non-executable memory (for storing data)

Memory protection

- Ensure processor (CPU) refuses to execute code in non-executable memory (won't run)
  - This can be done for the stack, or for arbitrary memory pages
- How does this help?
- Attacker can no longer jump to their own attack code, as any input they provide as attack code will be non-executable

Memory protection

- This solution is sometimes called "data execution protection" (DEP)

Limitations:

- Can still corrupt stack (change data)
- As long as return pointer points into existing code, memory protection will not block control transfer
  - Attacker can't add own code but can jump to things like the C library, try to run commands

Randomisation

- Address Space Layout Randomisation (ASLR)
- Attacker needs detailed information on memory layout (e.g., to jump to specific piece of code)
- So: randomise the layout every time we start a program
  - e.g., move the offset of the stack by some random value
- Attacker's job becomes much harder!

Randomisation

Limitations:

- Might not be possible to keep attacker from figuring out the address
- Some limitations on where objects can be located
- Brute force attacks or memory disclosures to map out memory on the fly
  - Disclosing a single address can reveal the location of all code within a library
- Hard to re-randomize once a program is running