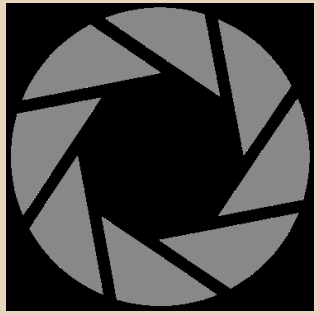


Fuzzing / Automated Testing

Offensive Computer Security
FSU CS Dept
Spring 2014

“Any sufficiently advanced bug is indistinguishable from a feature”

-Rich Kulawiec

Quoted in ch3 “Exploratory Software Testing” by James A Whittaker

Outline

1. Bugs
2. Testing
3. Fuzzing
4. CS Theory
5. Test Harness
6. More Fuzzing
7. Taint Analysis

Two types of testing

1. General Testing
    a. Regression testing
    b. developer written use cases
    c. spec-focused use cases
2. Random Testing
    a. fuzzing (The topic of this lecture)

fuzzing may find more bugs than all other forms of testing

Challenges of Testing

- How do we verify that the software performed correctly given arbitrary test cases?
  - Right output?
  - Side effects?
    - These rely on quality specifications

Challenges of Testing

- Can we distinguish bugs from features?
  - in product's specs / documentation???
    - if not, is testing impossible?
- If bug symptoms are so subtle that:
  - they evade automated testing
  - they evade manual testing…
    - is testing useless?

(Hopefully) Prior to Testing

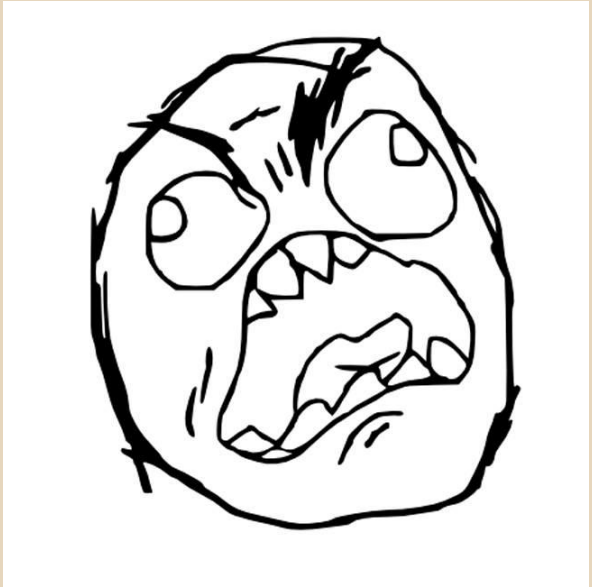
- Well documented code / written specs
- TDD (Testing Driven Development)
  - Test cases written by developers
    - Usually totally insufficient
- Security aware developers
  - So rare overall…
- Specifically explained testing expectations
  - non-existent. period.

In Reality

- All companies ship software that contains bugs
- Most testing is not concerned with security
  - across the spectrum / in general
- Quantity of bugs vs Quality of bugs found…
- Like security, Testing efforts aren’t appreciated if there is a major failure

In Reality

You will always have bugs that you just cant reproduce



A Honest Job Ad for a Tester

“Software tester wanted. Position requires comparing an insanely complicated, poorly documented product to a nonexistent or woefully incomplete specification. Help from original developers will be given grudgingly. Product will be used in environments that vary wildly with multiple users, multiple platforms, multiple languages, and other requirements yet unknown but just as important. We’re not quite sure how to define them, but security and performance are paramount, and post release failures are unacceptable and could cause us to go out of business”

QTD in “Exploratory Software Testing” by James A Whittaker

Discovering Vulnerabilities

Three Primary Methods:

1. **Source Code Auditing**
    - Requires source code
2. **Reverse Engineering**
    - Can be done without source code.
    - need binaries
    - hard
3. **Fuzzing**
    - Lots of tools / frameworks exist
    - Easy to make custom ones
    - Binary or source code availability is unimportant

Discovering Vulnerabilities

Three Primary Methods:

1. **Source Code Auditing (static)**
2. **Reverse Engineering (static)**
3. **Fuzzing (dynamic)**

There are other methods to keep in mind:

- Dynamic Taint Analysis (dynamic)
- Forward Symbolic Execution (dynamic)
- …

Discovering Vulnerabilities

Fuzzing primarily finds bugs.

- not all bugs are vulnerabilities.
- finding exploitable bugs.

What is fuzzing?

- **The (repeated) process of sending specific data to an application, in hope to elicit certain responses**
- **Specific?**
  - Mutated data, generational data, edge cases, unanticipated datatypes, etc.
- **Certain?**
  - crashes, errors, anomalous behavior, different application states...

Why?

Used *effectively* for:

- **Bug Hunting**
  - finding vulnerabilities (good guys & bad guys)
  - fame & profit (pwn2own ~$150k for first place)
- **Software testing (SDL)**
  - important to Google, Mozilla, Microsoft, Apple, etc.

Fuzzing Phases

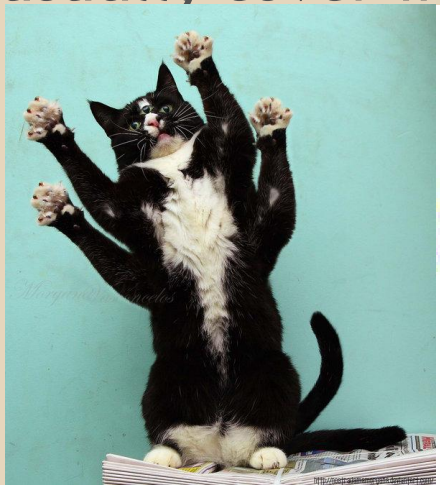
1. Identify inputs of application
2. Generate fuzzed data
    - We'll cover 3 methods
        i. Mutation
        ii. Generation
        iii. Differential
3. Execute Fuzzed Data
4. Monitor for Exceptions
5. Determine Exploitability

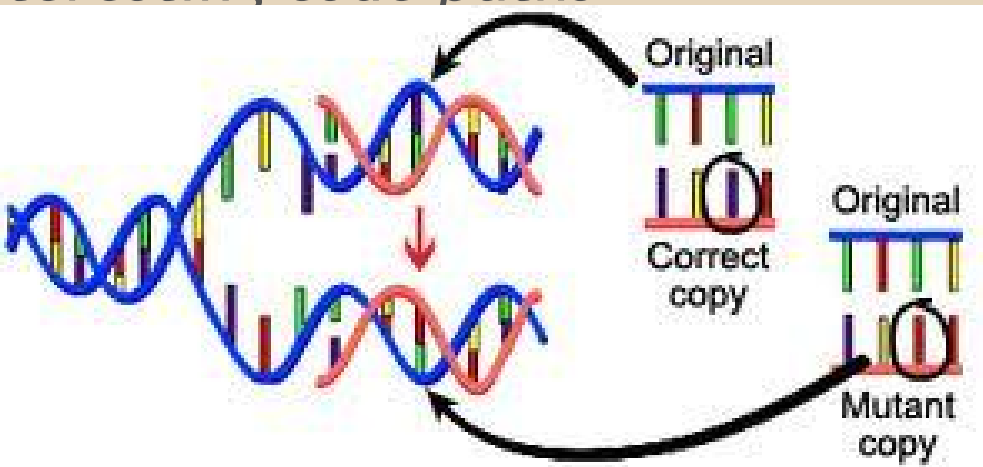
Methods for generating fuzzed data

- **Mutational fuzzing:**
  - starts with *known good "template" and seed* which is then modified (by the fuzzing algorithm).
  - Output is limited by the template and seed
    - anything that is NOT in the template or seed will not be generated
  - *i.e. take existing file and corrupt (mutate) parts of it and test application with it (over and over)*

Methods for generating fuzzed data

- **Mutational fuzzing limits:**
  - *fuzzing only as good as starting data samples*
    - *low entropy / complexity starting samples won't usually cover interesting code paths*



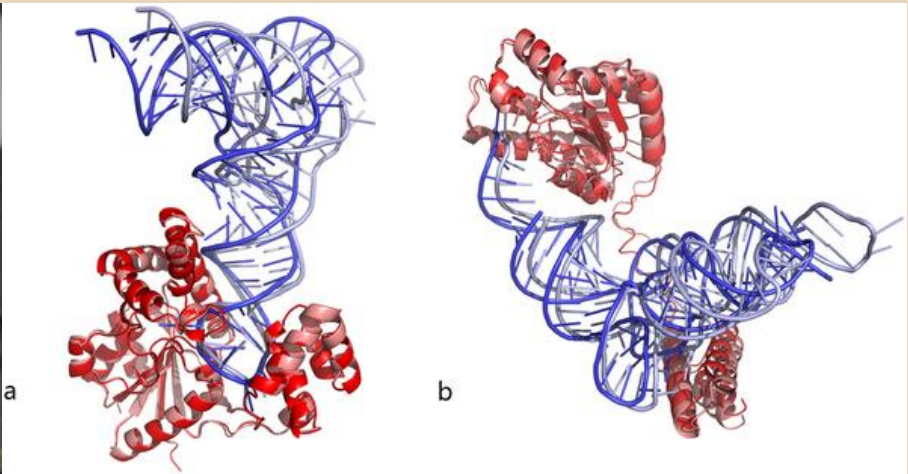


Methods for generating fuzzed data

- **Generational fuzzing:**
  - Capable of building the data being sent *based on data model* constructed by the fuzzer author
    - sometimes simple, dumb, or random
    - but can be highly efficient if written to combine good values in interesting ways
  - i.e. *you figure out the protocol / format and write code to generate it*.

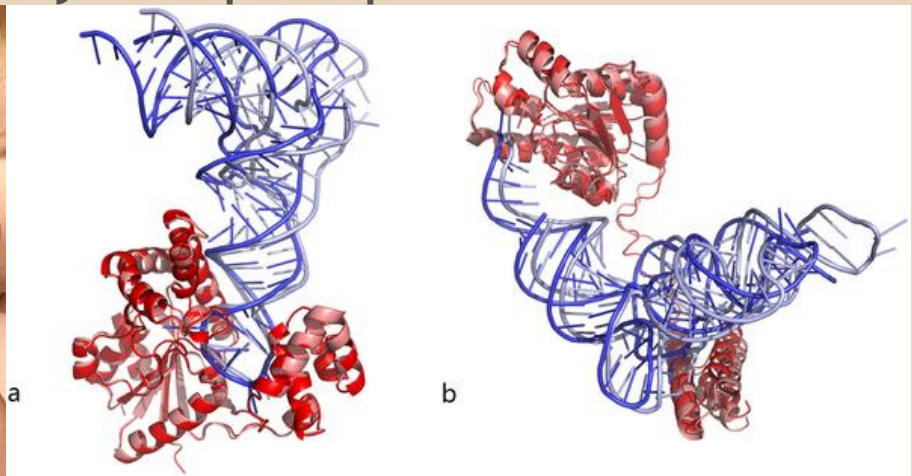
Methods for generating fuzzed data

- **Generational fuzzing limits:**
  - There are infinite unaccepted inputs to a program, and to each stage of a code path.
    - limit = your understanding of the input / constraints



Methods for generating fuzzed data

- **Generational fuzzing limits:**
  - Your generator is only as good a your understanding of the protocol / format
    - harder to generate very complex protocols



Methods for generating fuzzed data

- **Differential fuzzing:**
  - Any fuzzing algorithm that actively reduces the testing state space (other than plain exhausting it)
    - focuses on automating test-case reduction
    - focused on code path coverage
  - Trims the state space
    - constraint recognition
    - heuristics

The Eddington Number

1.57×10^{79}

(approx 10^{80})

Computer Science Theory

Space complexity

- if you try to generate and store > 10^{80}
  - Fail
- if you iterate through > 10^{80}
  - Possible & Slow + need algorithm to determine success
    - Crashes
    - Taint Analysis (later in this lecture)

Computer Science Theory

Space complexity Example

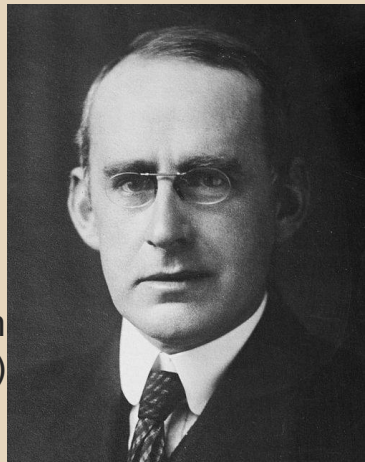
- HD Picture (1920x1080)
  - represented by html-friendly hex triplet:
    - byte 1: red values
    - byte 2: green
    - byte 3: blue
  - Each pixel represents 256^3 colors == 16,777,216.
- Fuzzing the whole space: $16{,}777{,}216^{(1920 * 1080)}$

Computer Science Theory

$1.50041... \times 10^{14981179} > 10^{80}$

- ➢ Some things are not feasible to **exhaustively** test or fuzz.
  - *thus there will ALWAYS be bugs*
    - for sufficiently large programs
- Important to target efforts.

Sir Arthur Eddington
(famous astrophysicist)



Computer Science Theory

Time Complexity

- Fuzzing is often parallelizable
  - huge help for dealing with time complexity
- If your fuzzer is $O(n^x)$ or significantly larger than O(n) you are probably doing it wrong.

A Fast File Fuzzer tool

http://rmadair.github.com/fuzzer/

- Python based mutational file fuzzer.
  - Uses PyDBG to monitor for signals of interest
- Client / Server architecture...

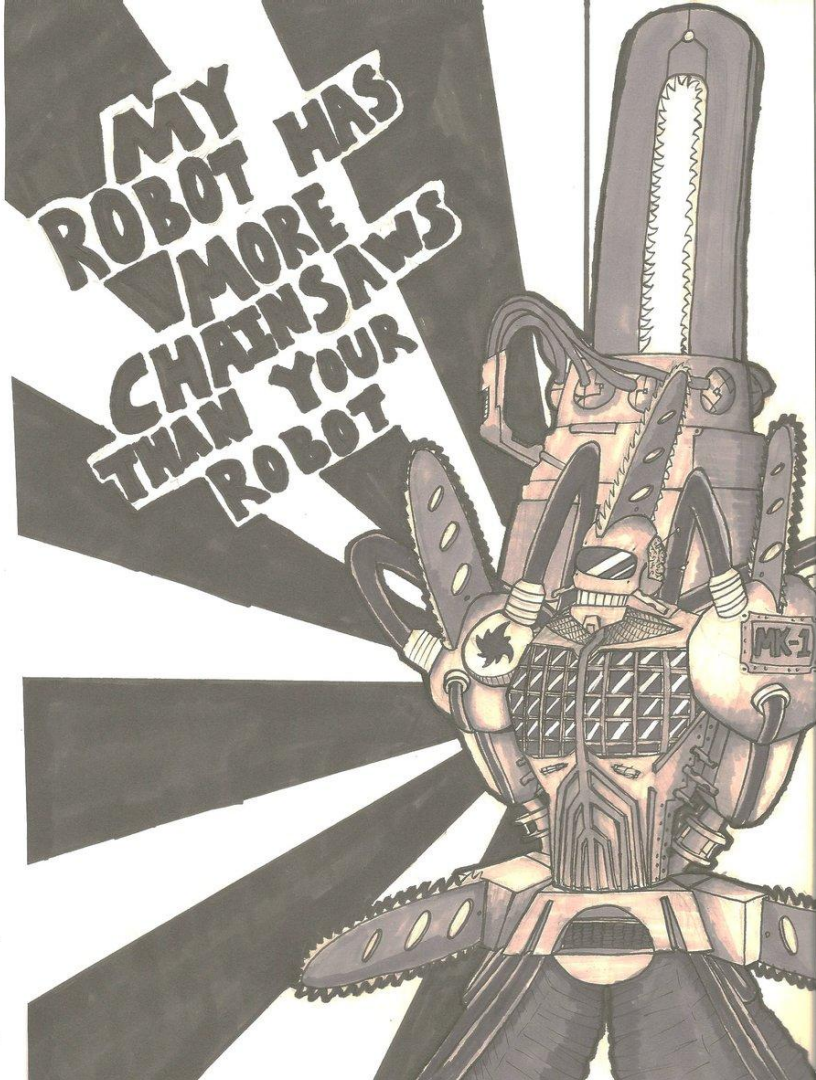
A Fast File Fuzzer tool

http://rmadair.github.com/fuzzer/

- Client / Server architecture
  - any number of clients can connect to the server
    - each client handles some portion of the fuzzing
      - creates mutated files clientside to fuzz a local copy of the target program with
  - can distribute fuzzing in a cloud like fashion
    - split up the set of all the things to fuzz over each client, and run them all in parallel

Recap (Fuzzing)

1. Mutational
2. Generational
3. Differential



Dynamic Analysis Basics / Fuzzing Test Harness

b/c fuzzing things without paying attention to them is a waste of time

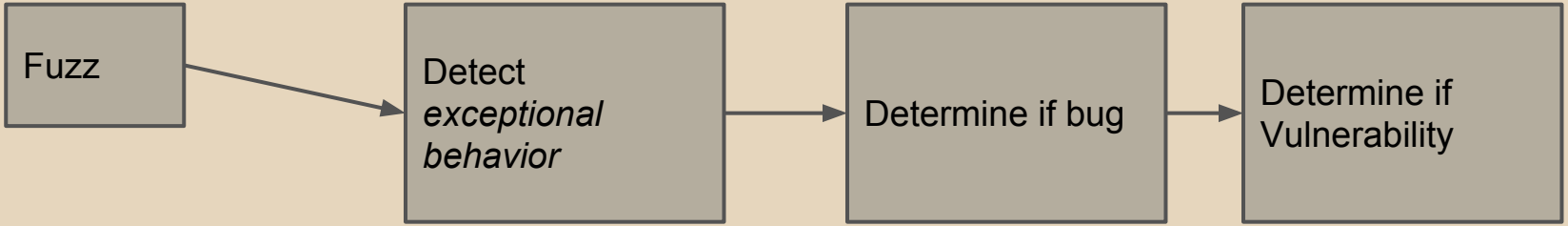
(white box) generating crashes

Crashes?

- Very easy to Detect

Logic Flaws?

- Very hard to detect. generally infeasible.



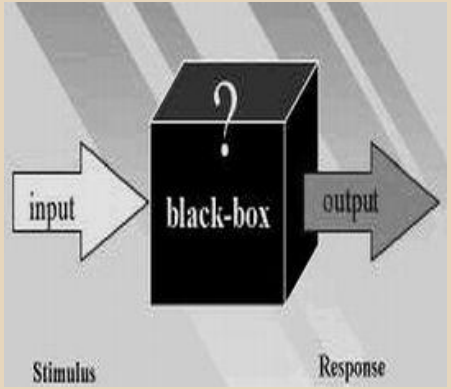
(white box) Test Harness

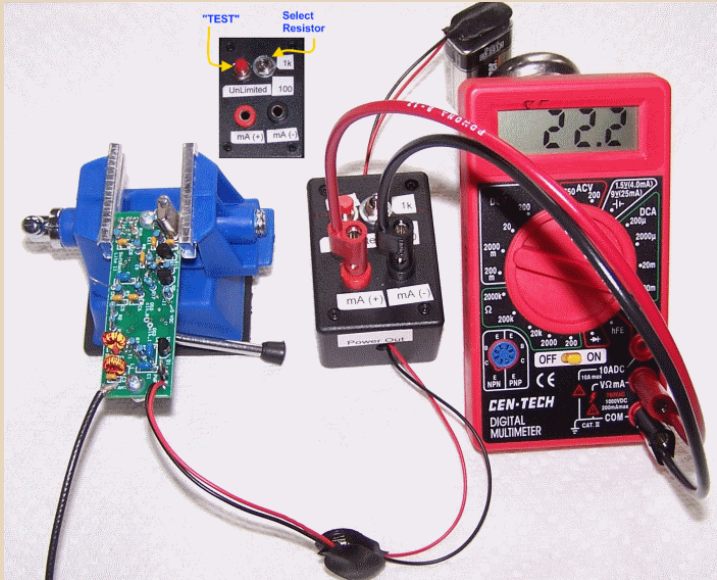
Test harness for crashes:

- check if process died
- check if process zombied
- check PID gone
- check logs
- attach debugger, check process state

(black box) Introspection

- Can you even get access of any form?
- Can you view it’s process state?
  - can you even view output?
  - can you even detect crashes?





Our homework

To build your own fuzzer and find bugs / vulns in a popular application

- then ethically disclose bugs / vulns to vendor

Guidance on Fuzzing

Fuzzing in General

The fuzzed input must be:

- *common enough* to pass elementary checks
  - i.e. basic constraints
- *uncommon enough* to trigger exceptional behavior

Constraints

```
if (x > 10U)
    //dangerous code
//safe code
```

Constraints

```
if (x ^ 0x012345 || strcmp(“SECRET KEY”, y))
{
    //path 1
} else {
    //path 2
}
```

BASIC CONSTRAINTS

```
int main(int argc, char* argv[]){
    if (argc != 9) {
        exit(1);
    }
    //rest of code
}
```

“QUALITY” Constraints

```
int foo(int x, char* data[]){
    if(x == complex_type){
        // do very complex operation on data[]
        …
    } else {
        // do really simple operation on data[]
    }
```

“QUALITY” Constraints

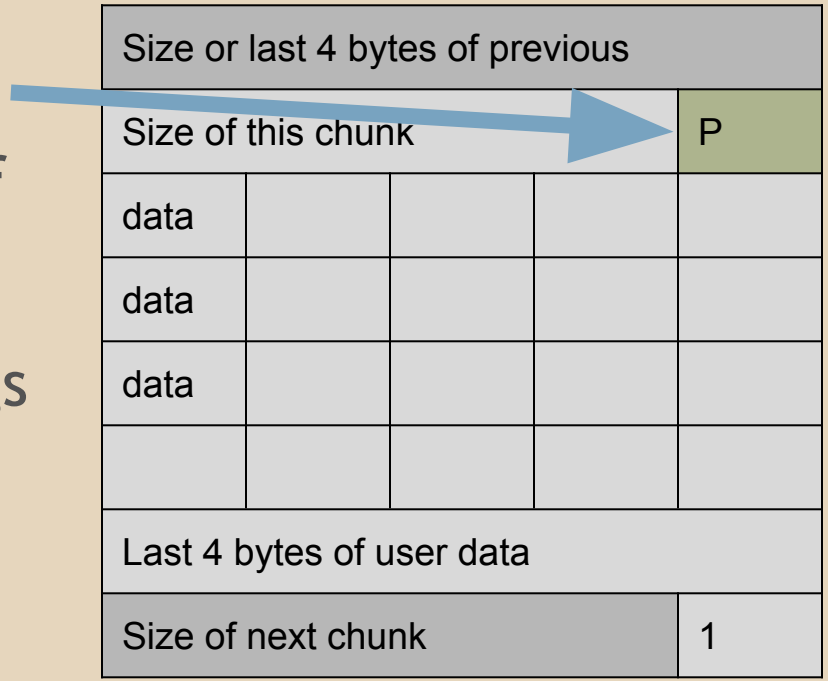
```
int foo(int x, char* data[]){
    if(x == complex_type){
        // do very complex operation on data[]
        ...
    } else {
        // do really simple operation on data[]
    }
```

KEEP THIS IN MIND AS WE GET TO CODE PATHS

Other "QUALITY" Constraints

Bit packing

- Remember how this influences the logic of free() / unlink?
  - Hard to detect logic bugs via fuzzing
- Frequent in very low level code



Types of Targets & Goals

- Environment Variables
- Positional Arguments, flags, etc.
- File formats
- Network protocols
- Web apps
- etc...

Types of Targets & Goals

Exploit/Attacker Goals:

- corrupt code/"business" logic
- Arbitrary/Malicious code execution
- permission escalation
- shell spawning / reverse shell
- etc..

5 properties to test (micro scale)

1. inputs
2. state(s)
3. code paths
4. user data
5. environment

5 properties to test (micro scale)

Independently fuzzing each is insufficient:

1. inputs
2. state(s)
3. code paths
4. user data
5. environment

Lets consider some examples

Thoughts on fuzzing

- Web Page: Add / Delete users (code path) may only be available to admin (state)
  - thus fuzz also flags/values in the cookie
- *Browsers/VMs/Java: Just in time (JIT) compilation and execution (code paths) heavily depend on global engine state*

Thoughts on fuzzing

- *Protocols:* Get protocol to a specific [state], then send unusual state change packets (data) to it

Testing Inputs

- 0,1,2,3… -1, -2 … ?
- 2, 4, 1024, 4096, …
- other atomic values?
  - practically infinite
- Or abstract inputs
  - Input length
  - Min / Max values and +/- edge case testing

Generating fuzzed data

What type of data should one fuzz an application with?

- **Integer values**
  - Border (edge) cases:
    - 0, 0xFFFFFFFF (2^32)
    - Leverage +n or -n cases
      - malloc (.... + 1)

Generating fuzzed data

- **Ranges:**
  - MAX32 – 16 <= MAX32 <= MAX32 + 16
  - MAX32 / 2 – 16 <= MAX32 / 2 <= MAX32 / 2 + 16
  - MAX32 / 3 – 16 <= MAX32 / 3 <= MAX32 / 3 + 16
  - MAX32 / 4 – 16 <= MAX32 / 4 <= MAX32 / 4 + 16
  - MAX16 – 16 <= MAX16 <= MAX16 + 16
  - MAX16 / 2 – 16 <= MAX16 / 2 <= MAX16 / 2 + 16
  - MAX16 / 3 – 16 <= MAX16 / 3 <= MAX16 / 3 + 16
  - MAX16 / 4 – 16 <= MAX16 / 4 <= MAX16 / 4 + 16
  - MAX8 – 16 <= MAX8 <= MAX8 + 16
  - MAX8 / 2 – 16 <= MAX8 / 2 <= MAX8 / 2 + 16
  - MAX8 / 3 – 16 <= MAX8 / 3 <= MAX8 / 3 + 16
  - MAX8 / 4 – 16 <= MAX8 / 4 <= MAX8 / 4 + 16

Try to influence signed / unsigned values: char short, int, long, etc.

Unsigned value:
2^X

Signed value:
2^X / 2

Generating fuzzed data, cont

- **String repetitions:**
  - A*10, A*100, A*1000
    - *$./program $(perl -e 'print "A" x1000')*
    - *$./program $(python -c 'print "A"*1000')*
  - Not just 'A', 'B' makes a difference on the heap, and in hard coded anti-reversing checks!
    - *like in CTFs*

Generating fuzzed data, cont

- **Delimiters**
  - !@#$%^&*()-_=+{}|\;:’”,<.>/?~`
  - Varying length strings separated by delims
  - increasing length of delimiter:
    - *User*::::::::::::::::::::::::::::*password*

Generating fuzzed data, cont

- **Format Strings**
  - %s and %n have greatest chance to trigger a fault
    - %s dereferences a stack value
    - %n writes to a pointer (another dereference)
  - Should fuzz long sequences (i.e. to cause crashes)

Generating fuzzed data, cont

- **Character translations**
  - 0xfe and 0xff are expanded into 4 characters under UTF16
  - 0xcc and 0xcd modifiers super and sub accents for UTF8 extended encodings:

Generating fuzzed data, cont

- Character translations
  - for instance: 
  - unpacked and decoded in python, this is:
    'U','\xcd','\xab','\xcc','\x81','\xcd','\x97','\xcd','\x86','\xcc','\xbd','\xcc','\x88','\xcc','\x86','\xcd','\x9e','\xcc','\xb1','\xcc','\xb2','S','\xcc','\x8a','\xcc','\x8c','\xcd','\xae','\xcc','\x88','\xcc','\x80','\xcd','\x83','\xcc','\x88','\xcc','\xa7','\xcd','\x87','\xcc','\xbc','\xcc','\x9c'
  - see http://www.utf8-chartable.de/unicode-utf8-table.pl?start=768&number=128&names=-&utf8=0x

Generating fuzzed data, cont

- **Directory Traversal:**
  - targeting web apps,network daemons, etc
  - ../../ and ..\..\ etc...
    - *important to try different character encoding (%5C = '\' in unicode)*

Generating fuzzed data, cont

- **Metacharacter / Command Injection**
  - when targeting web apps, cgi scripts, network daemons
  - &&, ; --' “ ” , > < ! % $() and | characters

Generating fuzzed data, cont

- File types
  - spoof magic number (unix)
    - 2-byte identifier at the beginning of a file
    - .gif's have magic numbers of GIF87a or GIF89a
  - spoof file extension
    - old file extension types (i.e. .php3 instead of .php)
  - content-meta data (in web traffic)
    - i.e. via intercept proxy
  - special folders (windows mainly)

Poly-File Types

http://code.google.com/p/corkami/downloads/detail?name=CorkaMIX.zip&can=2&q=

Proof of Concept to generate a file that is a valid PE, PDF, HTML (+ java script), AND .JAR (with Python) file!

Generating fuzzed data, cont (Networking)

- **Modeling Arbitrary Network Protocols**
  - What if SMTP or a proprietary protocol is tunneled over HTTP to your web app?
    - or over SSH
    - or over DNS
- **Bit flipping for protocol headers / flags**
- **Fuzz with network time syncing protocols**
  - perhaps to attack crypto on a network service **:D**
  - in use since 1985

Generating fuzzed data, cont (Networking)

- Modeling Unknown Network Protocols
  - Turns out that Bio-Mathematical pattern mapping techniques work quite well for detecting structures of protocols
    - very helpful for modelling them, building state machine, and generating the protocol given sample traffic
      - See Offensive Network Security for more!

Crash Analysis / Taint Analysis

Once crashes are caused, determining if a vulnerability exists

Taint Analysis

- Goal is to mark data originating from untrusted sources as *tainted*
- can be done statically / dynaically
- Two dependencies that determine taint:
  - Data flow dependencies
  - Control flow dependencies

Data flow dependencies

```
//x is tainted
 y = 2;
 z = x + y;
//z is tainted
```

From: http://diyhpl.us/~bryan/papers2/paperbot/e15bb28f0dc692c053f64bb48b879ab3.pdf

Control flow dependencies

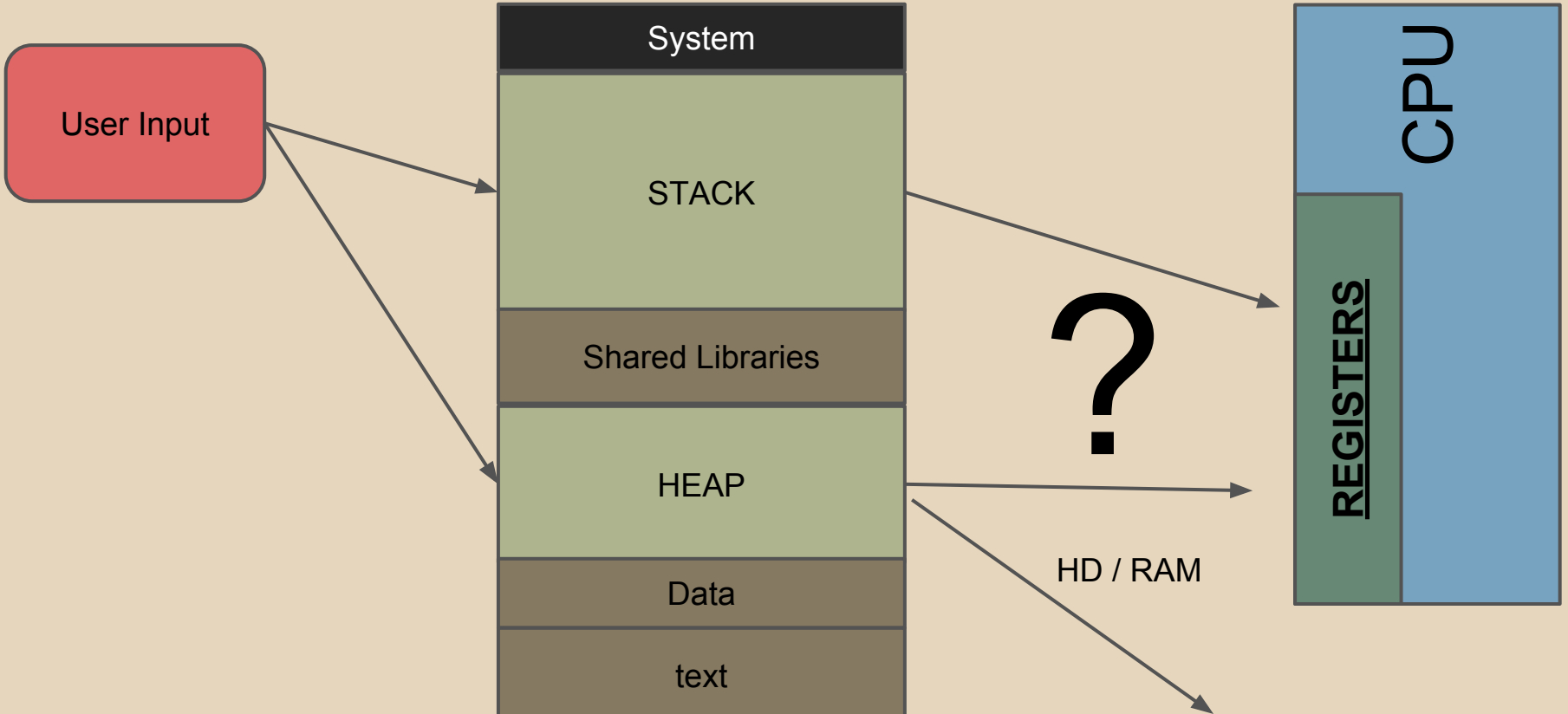
```
//x is tainted
if (x > 1) y = 1 else y =2;
//y is tainted b/c influenced by x
```

From: http://diyhpl.us/~bryan/papers2/paperbot/e15bb28f0dc692c053f64bb48b879ab3.pdf

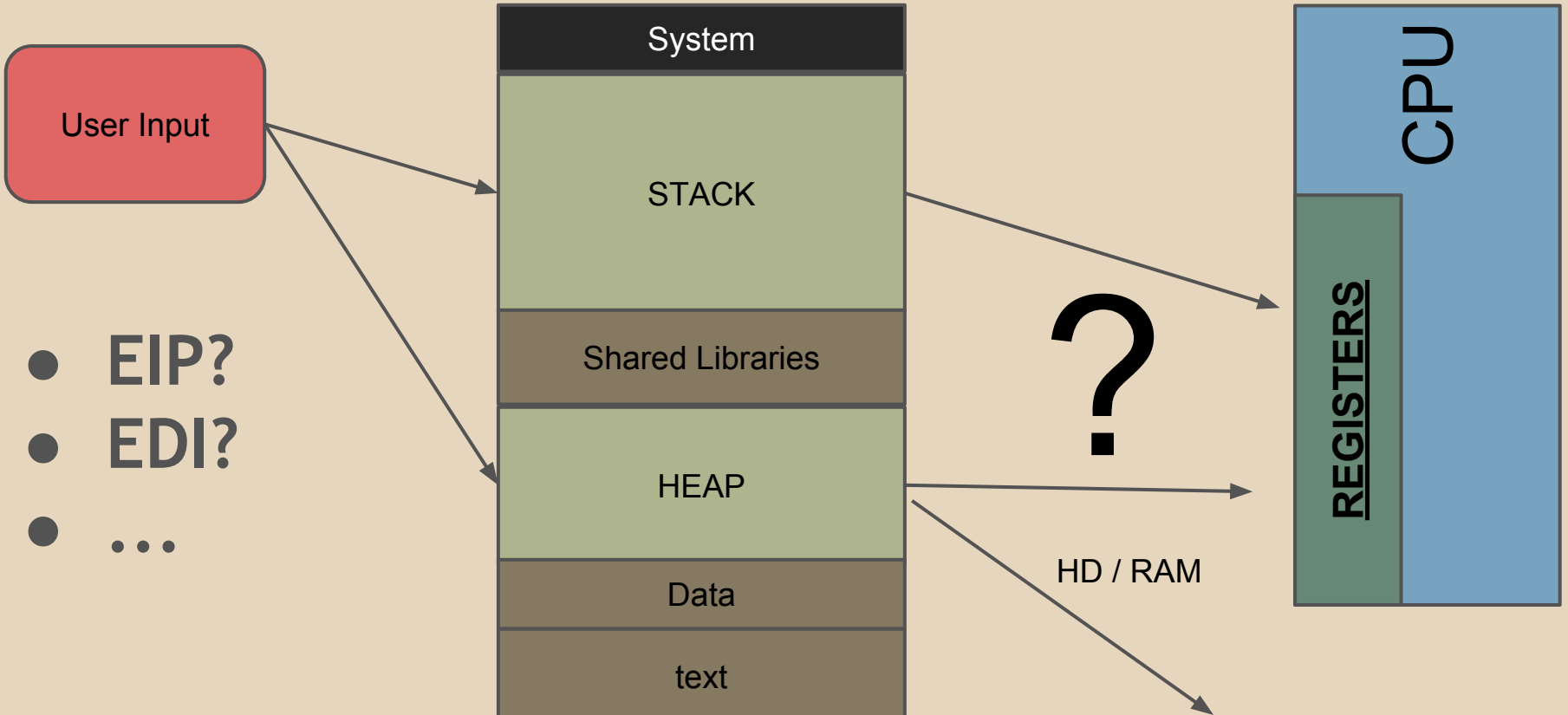
Taint Tracking Policies

- Just track data flow dependencies?
- Also data flow dependencies?
- Track taint after free() / garbage collection?
  - miss use after free or use uninitialized vulns.
- bytewise?
  - or even bitwise?

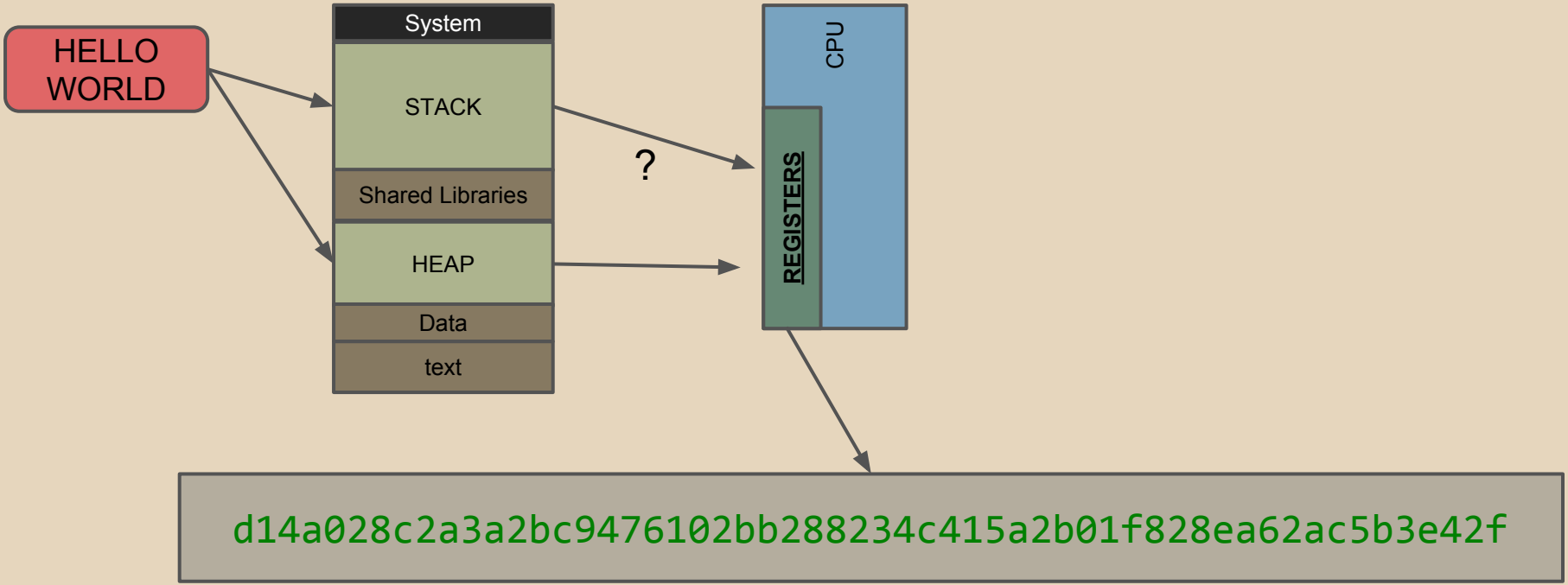
Taint Analysis



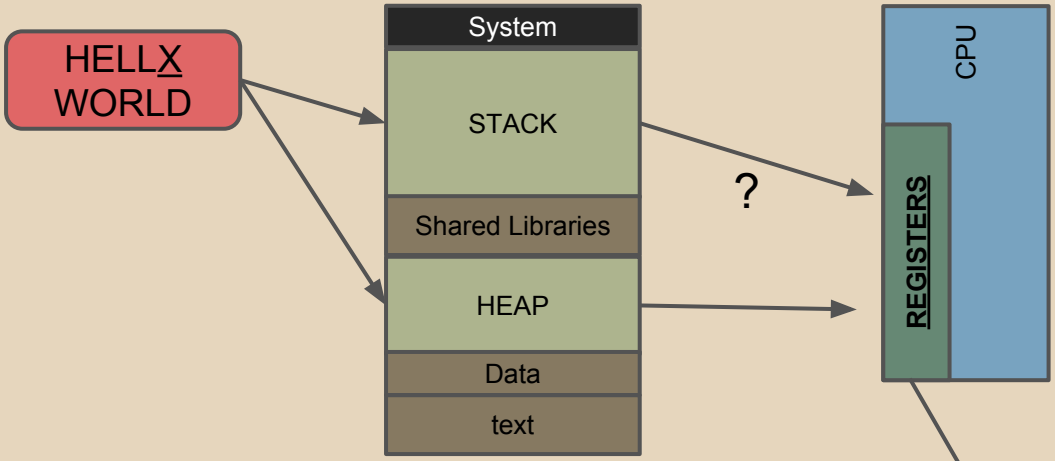
Taint Analysis



Taint Analysis



Taint Analysis
(For Differential Cryptanalysis)



This is a toy example

d14a028c2a3a2bc9476102bb288234c415a2b01f828ea62ac5b3e42f

Taint Analysis

- Partial writes / Full writes
- operations like shifts, and, nand, ...
  - ones that destroy information

Stop tracking taint when

- variable overwritten by static /const value
- var assigned from untainted ojbect

Simple Taint Analysis

User inputs are non-repeating patterns:
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5...

Simple Taint Analysis

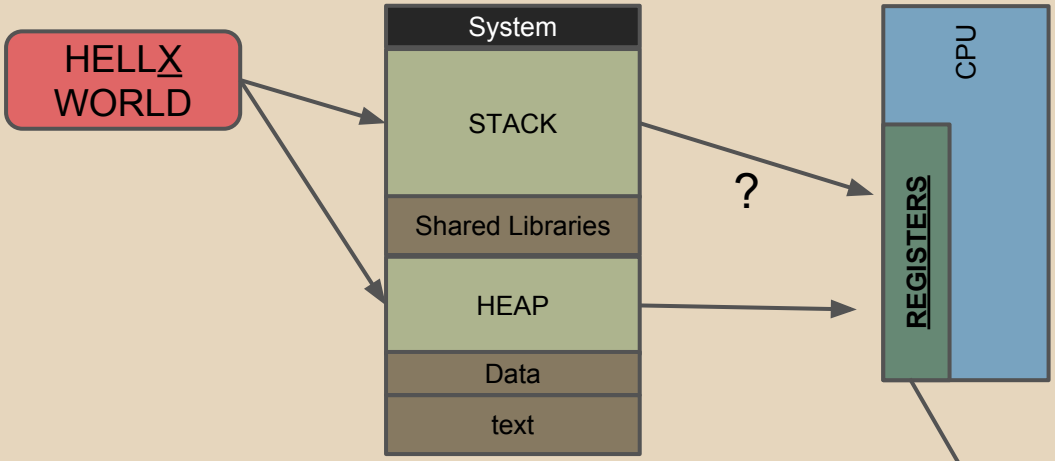
Check if registers at crash contain any of these patterns(ENDIANNESS MATTERS HERE!)

EIP = 5Af6 (big endian) f65A (little endian)

RIP = Ag1Ag2Ag (big endian)
Agg21AAg (little endian (x86))

If confused, see (http://en.wikipedia.org/wiki/Endianness)

Taint Analysis
(For Differential Cryptanalysis)



This is a toy example

d14a028c2a3a2bc9476102bb288234c415a2b01f828ea62ac5b3e42f

Problem

- Patterns won’t work in all applications
- method fails on any transforms of user input
- encoding / decoding / expansion /etc…

Questions?

MORE NEXT TIME!!

Read: Differential Testing for Software (http://www.cs.dartmouth.edu/~mckeeman/references/DifferentialTestingForSoftware.pdf)
Read Adaptive Random Testing (http://www.utdallas.edu/~ewong/SYSM-6310/03-Lecture/02-ART-paper-01.pdf)
Read: Attaching the Rocket to the Chainsaw https://www.cert.org/blogs/certcc/2013/09/putting_the_rocket_on_the_chai.html

http://blog.regehr.org/archives/1039

BFF

Basic Fuzzing Framework (Linux / Mac)

http://www.cert.org/vulnerability-analysis/tools/bff.cfm

FOE

Failure Observation Engine (Windows fuzzing)

http://www.cert.org/vulnerability-analysis/tools/foe.cfm