Fuzzing

Fuzzing

- A way to find input-parsing bugs by randomly or systematically modifying input streams
- Can be random (no knowledge of input formats), smart (handles input formats, checksums, etc.), black box (smart, but with no validation of code coverage), or white box (systematically test different code paths)
- Input data can be generated automatically or by mutating valid inputs
- Extremely powerful technique, used by testers and attackers

Why Does Fuzzing Work?

- It exercises seldom-tested code paths
- It pushes boundary conditions
- Note: tester must use other tools to look for memory leaks, deadlocks, code coverage, etc.

Who Fuzzes?

- Developers
- Security testers
- Quality assurance groups
- Attackers. . .

Security Scanners versus Fuzzers

- Scanners are reactive—they only find known problems
- Fuzzers can find unknown problems
- (Similar issue with regression testing)

Fuzzing Isn’t Ordinary Testing

- Ordinary testing checks if code meets the requirements
- “Input lines longer than 512 characters must be rejected”
- In other words, it finds knowable issues—and possibly not correctly
- A test suite might have a 513-character line, which the program rejects—but perhaps a 5120-character line will cause a buffer overflow before that check happens

The Linux Man Page for `gets()`

> ***gets()*** *returns s on success, and NULL on error or when end of file occurs while no characters have been read. However, given the lack of buffer overrun checking, there can be no guarantees that the function will even return.*

In other words, a `gets()` read into a 513-byte buffer can be used for 512 bytes or to detect 513 bytes—but a longer input line may never reach the check.

Goals: Testing versus Fuzzing

- Testing: Find which requirement isn’t met
- Fuzzing: Generally, crash the program under test
- Fuzzing is not about finding vulnerabilities per se, but most fuzzing-induced crashes can be turned into exploits

Fuzzing Usually Isn’t Random Input

```
$ nc www.cs.columbia.edu 80 </dev/urandom
HTTP/1.1 400 Bad request
content-length: 90
cache-control: no-cache
content-type: text/html
connection: close

<html><body><h1>400 Bad request</h1>
Your browser sent an invalid request.
</body></html>
```

By Contrast

```
$ nc eu.httpbin.org 80
GET /fa4bfb50d9e3e826860eecbd86d623e7cddb HTTP/1.0

HTTP/1.1 404 NOT FOUND
Date: Tue, 14 Apr 2020 18:02:35 GMT
Content-Type: text/html
Content-Length: 233
Connection: close
Server: gunicorn/19.9.0
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>404 Not Found</title>
<h1>Not Found</h1>
<p>The requested URL was not found on the server. If you entered the
URL manually please check your spelling and try again.</p>
```

Which is More Likely to Find Problems?

- Simple nonsense is easy to reject—for HTTP, just see if the first few characters match a valid command
- Syntax-driven fuzzing can find deeper problems
- Better yet: follow the state machine

Fuzzing and Protocol States

- Protocols often have different states
- You can't test behavior in a later state if you don't reach it successfully, generally via valid inputs
- Example: you can't fuzz an HTTPS-only website if you never successfully negotiate the TLS connection

Fuzzing Strategies

- Knowledge of generic input is necessary
- Two broad types: *generation* and *mutation* fuzzers
- Both have their uses

Generation Fuzzers

- Typically start with a grammar (perhaps in BNF) of the accepted language
- Generate random inputs that match that grammar
- Include special notation for, e.g., length fields, checksums, etc.
- Note: obviously, grammars are different for binary protocols than for ASCII or Unicode

Mutation Fuzzers

- Provide samples of valid inputs
- The fuzzer generates changes to the valid input
- Again, there are special provisions for things like length fields
- Of course, invalid length fields are also interesting. . .

Hybrid Approaches

- Some fuzzers do both
- Often: use one approach to get to some protocol level, then switch to another
- Some use libraries of known troublesome patterns, e.g., filenames with `/../../../../etc/passwd` in them
- Sometimes, this is a good spot for random inputs

When Fuzzing Fails

- Suppose, over the history of a project, you've been fuzzing a module, and finding and fixing bugs
- One day, fuzzing no longer crashes it
- Is it now (security) bug-free? Or are there bugs that the current fuzzer can't find?

The Pesticide Paradox

- Any testing method leaves a residue of subtle bugs it couldn't find
- This lets the complexity of the code grow
- We've eliminated the easy bugs, leaving subtle ones
- Conclusion: when fuzzing doesn't find bugs, fuzz harder!

Black Box, White Box

- Black box fuzzing: go after the binary, with no knowledge of the source code
- White box: use the source to guide fuzzers

☞ Can measure code coverage

☞ Can test using hidden or undocumented parameters

Instrumentation-Guided Fuzzers

- Add instrumentation to your code to tell the fuzzer what has happened
- Better yet, have your compiler add the instrumentation
- The fuzzer will try to avoid paths it has already exercised and look for new ones
- Result: more code coverage testing
- N.B.: the longer a fuzzer runs and the more code paths it has to cover, more likely it is that the program has a high attack surface
- Similar conclusions if the fuzzer finds many different classes of bugs

Levels of Fuzzing

- Individual routines, via unit test frameworks or in-memory changes
- Single programs
- Network APIs
- Files
- All the myriad ways that web servers can fail

History of Fuzzing

- Some early work in the 1970s and 1980s
- (As an undergraduate, my friends and I would sometimes feed object files to a compiler, to see how crazily it would react, but we had no deeper motive)
- Early testing of TCP/IP—you could get points for “KOing your opponent”, i.e., crashing another implementation
- More formal use in software testing in the 1990s; some academic interest
- 1999–2001: Oulu University, Finland

Oulu University

- In 1999, they built the PROTOS fuzzer, which they used to fuzz many important network protocols
- They found many flaws, most importantly in SNMP
- SNMP: Simple Network Management Protocol
- Used by ISPs and many enterprises to monitor routers and other network elements
- *Everyone* had to patch on short notice. . .
- This put fuzzing on the map—and since some of the problems found were security problems, it put fuzzing on the security map

Why SNMP?

- SNMP packets are defined using ASN.1 (Abstract Syntax Notation 1)
- ASN.1 is very complex; the field definitions are translated into C by a compiler
- Many implementations used the same compiler. . .

An ASN.1 Example from SNMP

```
GetRequest-PDU ::=
      [0]
          IMPLICIT SEQUENCE {
                  request-id
                      RequestID,

                  error-status          - always 0
                      ErrorStatus,

                  error-index           - always 0
                      ErrorIndex,

                  variable-bindings
                      VarBindList
          }
```

Why Fuzzing?

- More precisely, why am I talking about fuzzing in this class, which focuses on *system* security?
- What do we fuzz?
- Why?
- What do we do with the answers?

What is the Lesson of the SNMP Problem?

- ASN.1 is complex; therefore, there is (probably) a large attack surface
- There was a common mode failure: a single compiler
- The defenses either failed or weren’t deployed

SNMP Authentication via the “Community” String

```
COMMUNITY-BASED-SNMPv2 DEFINITIONS ::= BEGIN
- top-level message
    Message ::= {
            SEQUENCE {
                 version
                    INTEGER {
                        version(1)   - modified from RFC 1157
                    },
                community           - community name
                    OCTET STRING,
                data                - PDUs: SNMP commands
                    ANY
            }
    }
END
```

What’s the Problem?

- The community string—the password—is encoded in ASN.1, too
- The compiled code has to parse the over-the-wire ASN.1 to extract and then validate the password
- But the parser was buggy. . .
- That is: the authentication mechanism was in a module with a very high attack surface

☞ Fuzzing exposed, as an implementation issue, what should have been clear as an architectural point

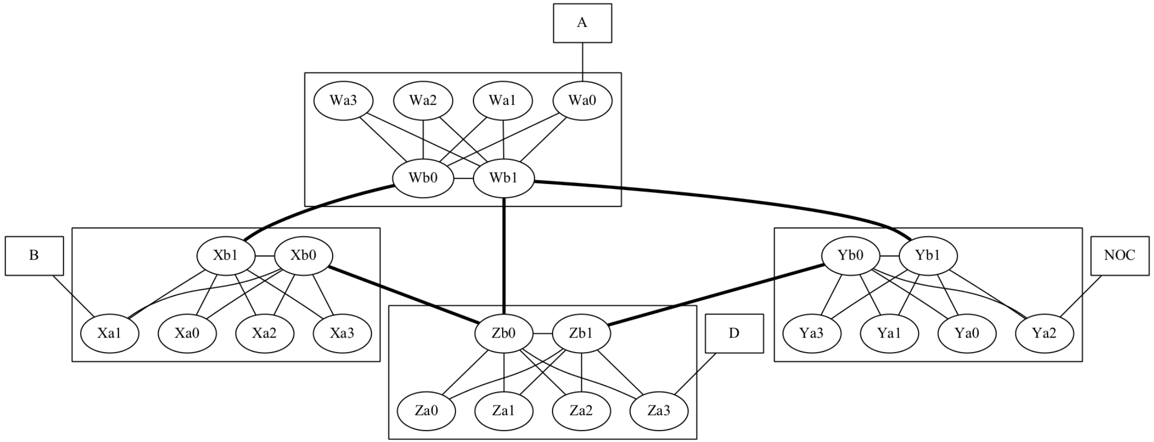
(We Could Have Lost the Internet!)

- This was a critical vulnerability; ISPs had to patch or install filtering immediately
- But—major ISPs test new vendor software releases for *months* before putting them into production
- Even installing packet filters is risky—you can accidentally lock yourself out

Suggested Mitigations

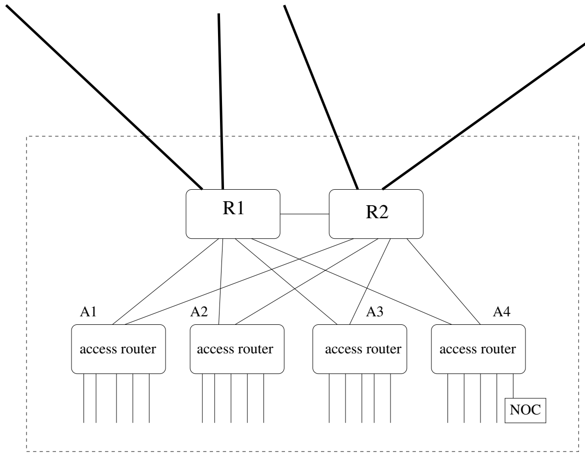
- The mitigations that CERT suggested are also interesting
- Disable SNMP—but ISPs can’t
- Ingress filtering—block SNMP packets from outside the ISP network
- Block the SNMP port from unauthorized internal hosts—good idea for enterprises, but not very applicable to ISPs
- (And what about zero-trust architectures?)
- Put SNMP on a separate management network only
- What do these tell us?
- Let’s look at an ISP topology

(Simplified) ISP Backbone



POPs are connected by backbone links

Point of Presence (POP)



Customers and peers (other ISPs) are connected to *access routers* A1, A2, A3, A4; inter-POP links are via *backbone routers* R1 and R2

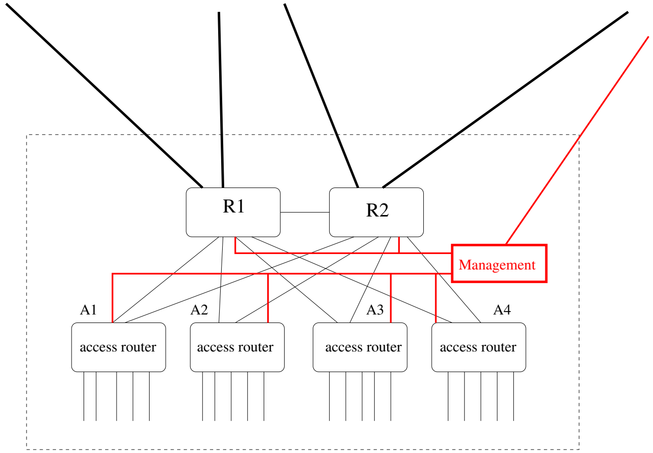
Managing this Network

- How does the NOC connect?
- Primarily in-band: over the Internet
- Is this wise?
- And what about after the SNMP fuzzing result?

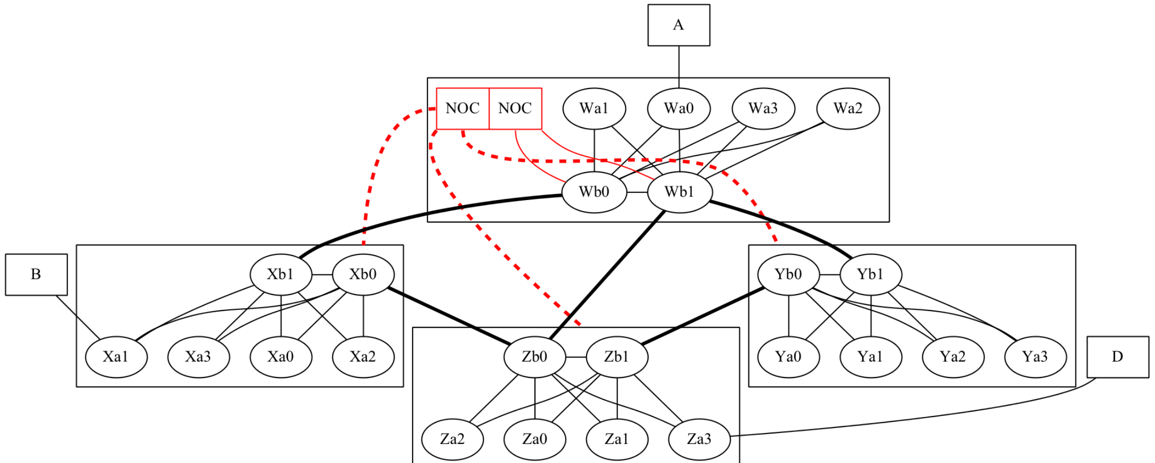
A Management Network

- Add a separate management network
- Only allow SNMP over it
- But what about reliability?

A POP with a Management Network



Managed Backbone



Reliability

- ISPs *require* reliability
- They *must* have the ability to manage all of their devices
- Single points of failure—like a management network or a special management router—are not acceptable
- ISPs do have management networks—but they're backups to over-the-Internet management, which provides two independent paths to every element
- We have to secure the Internet path!

Securing Many Routers

- Suppose you need to push out filters to many, many routers. How do you do this?
- This is only feasible if you've prepared in advance, if you have tools to manage all of your endpoints
- You can't possibly hand-edit thousands of routers' configuration files
- Tool-building always pays for for sysadmins. . .

Design for Management

- For enterprises, you have to design things to be managed
- System management has many different pieces: monitoring network elements and servers, pushing configuration changes, patching software, diagnosing and rebooting machines, tracking which laptops haven’t been patched, etc.
- Example: how do you secure access to a power bar, a device that lets you power-cycle a computer remotely? What if it doesn’t support your preferred style of authentication?

Where Else is ASN.1?

- If SNMP was vulnerable because of ASN.1, what else uses ASN.1?
- TLS, S/MIME, anything involving X.509 certificates
- Why? Because they were dealing with complex, over-the-wire data structures, with varying precision integers, varying length strings, optional fields, and more
- This all has to work on many different computer architectures
- *Some* form of binary field description language is needed
- Also: the IETF decided to use OSI's X.509 certificate format; it was defined in ASN.1
- (There's a long political story about that, too)

Dealing with ASN.1 Software

- Anything that uses ASN.1 can be presumed to have a high attack surface
- How do we handle this?
- Can we
    1. Secure ASN.1?
    2. Stop using ASN.1?
    3. Isolate it?
    4. Something else?
- None of the choices are great. . .

Securing ASN.1?

- Nope!
- Remember the actual problem statement: not that it's insecure, but rather, that it has a high attack surface
- We can try to audit the code, test the code—and fuzz the code
- Architecturally, though, we want to move it out of critical locations

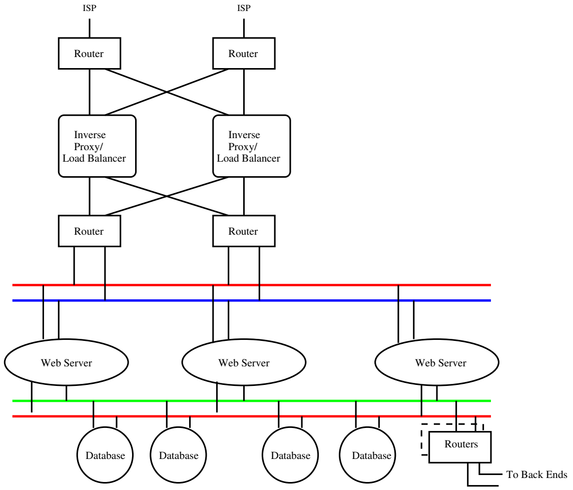
Stop Using ASN.1?

- Not possible—too many essential services rely on it
- You can't run a web site without TLS
- You can't run enterprise-grade router complexes without SNMP
- Many VPNs require X.509 certificates

Isolation?

- Sometimes, we can isolate ASN.1 software
- More precisely, we can parse it where failures are less harmful
- But this isn't always possible

Isolating TLS



- We can terminate TLS on the load balancers or on the web servers
- If a load balancer is hacked, active traffic can be sniffed and the web server is more open to attack
- If a web server is hacked, the attacker has direct access to the databases—far more serious!
- Conclusion: it's safer to terminate TLS on the load balancer

Google’s Access Proxies

- Access proxies also terminate TLS connections
- If it was hacked, the web service behind it would be exposed
- Access proxies can attach login information—which could be spoofed
- How many web services does a typical AP serve? That is, how many services are exposed to a single AP failure?

VPNs

- VPN gateways handle encryption and decryption—but they’re also access control points
- If one is hacked, anyone can get into the network
- If we offloaded the certificate processing and that machine was hacked, it wouldn’t validate certificates properly, so attackers could still get in
- Conclusion: we cannot isolate a VPN gateway; all we can do is monitor it

Fuzzing

- Fuzzing should be part of every test group and every security group's toolkit
- Using it properly is *work*
- If properly used, it gives guidance on attack surface as well as on bugs
- System architects need to know how to use the output

Questions?

(Palm warbler, Central Park, April 2, 2022)