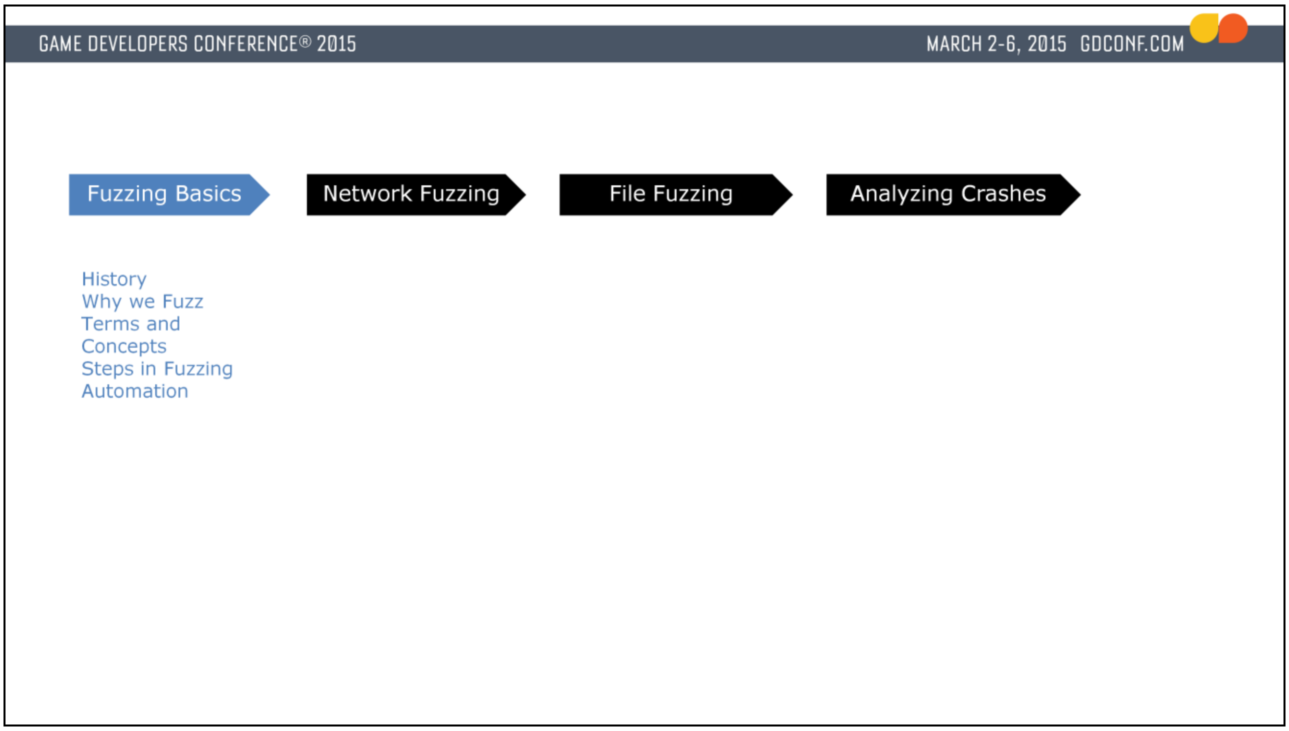
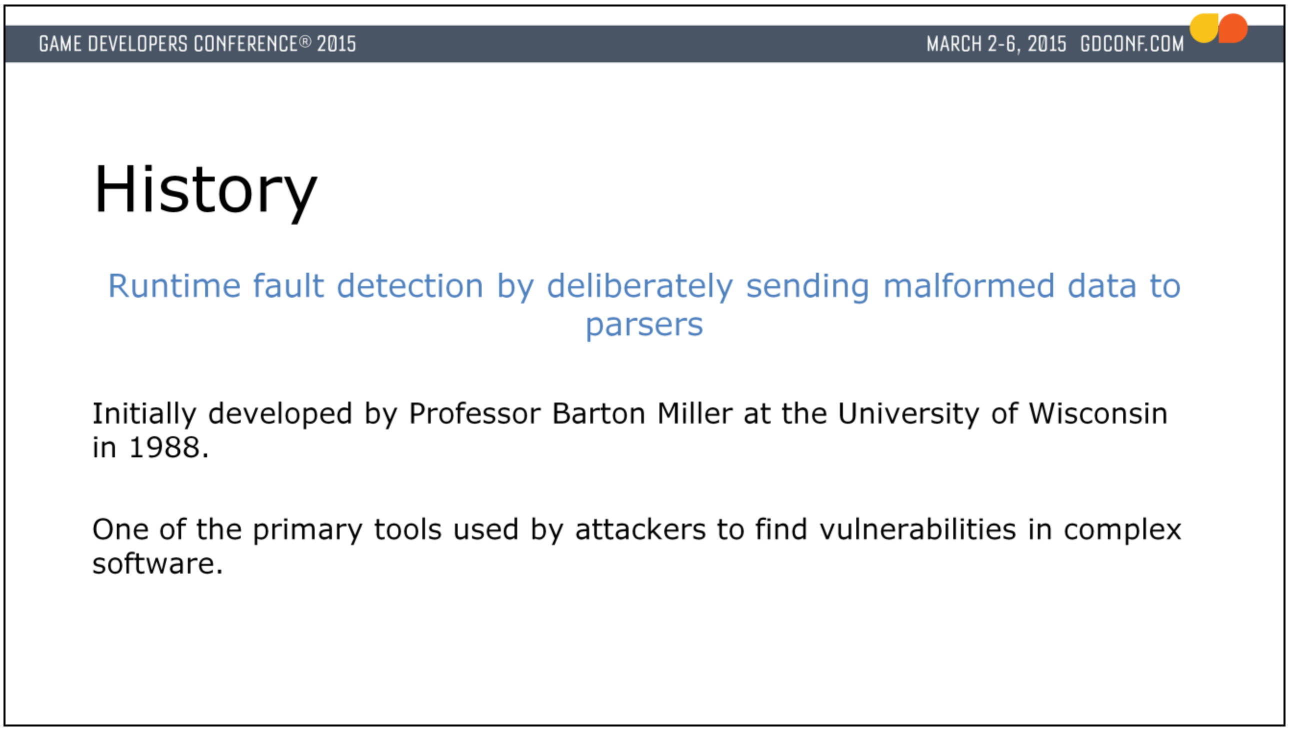
How to Protect Your Game's Code With Fuzzing

Dave Weinstein
Security Researcher, HP Security Research

GAME DEVELOPERS CONFERENCE®
MOSCONE CENTER · SAN FRANCISCO, CA
MARCH 2-6, 2015 · EXPO: MARCH 4-6, 2015




GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

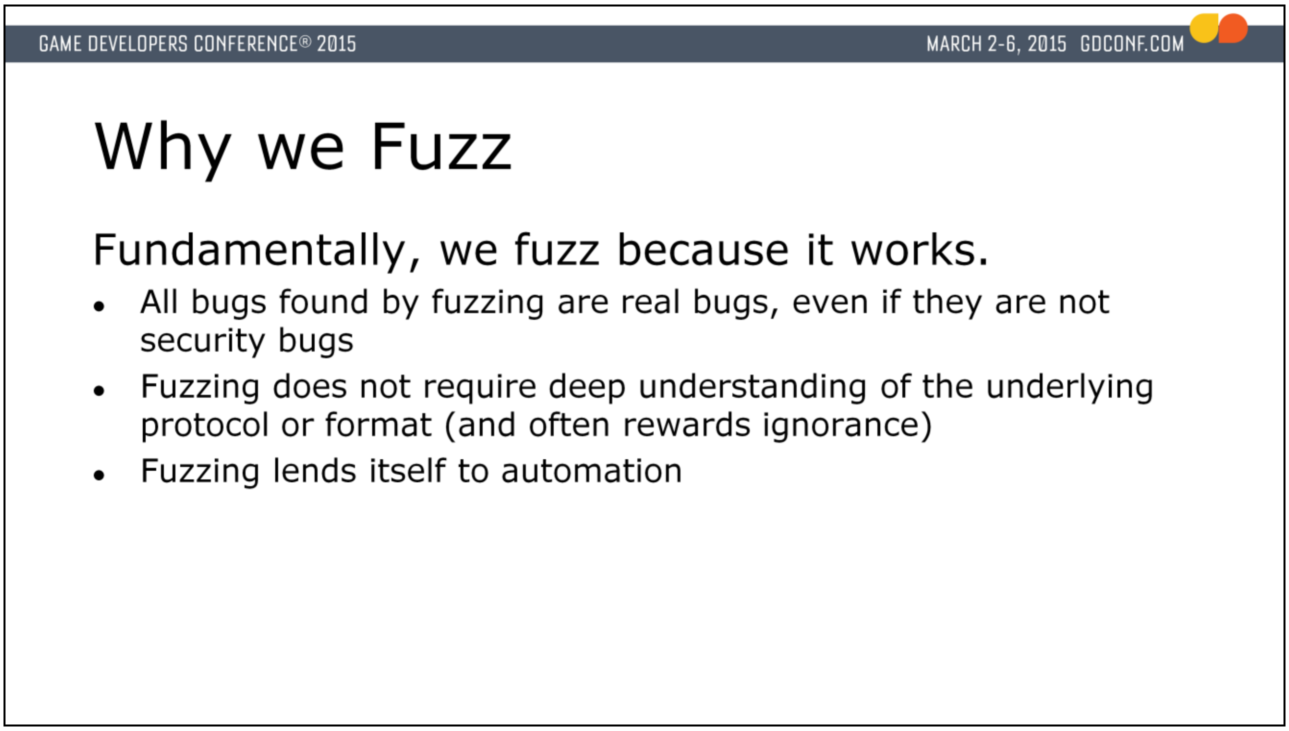
History

Runtime fault detection by deliberately sending malformed data to parsers

Initially developed by Professor Barton Miller at the University of Wisconsin in 1988.

One of the primary tools used by attackers to find vulnerabilities in complex software.

It was a dark and stormy night. Seriously. There was a rain storm in Wisconsin, and the line noise dialing into the Unix machines was bad enough to keep putting garbage characters into the command line arguments. And the tools, rather than giving an error message, crashed. And that was the start of fuzzing. Fuzzing is enormously effective. If your software is successful enough, someone will fuzz it. It would be nice if that someone was you.



Why we Fuzz

Fundamentally, we fuzz because it works.

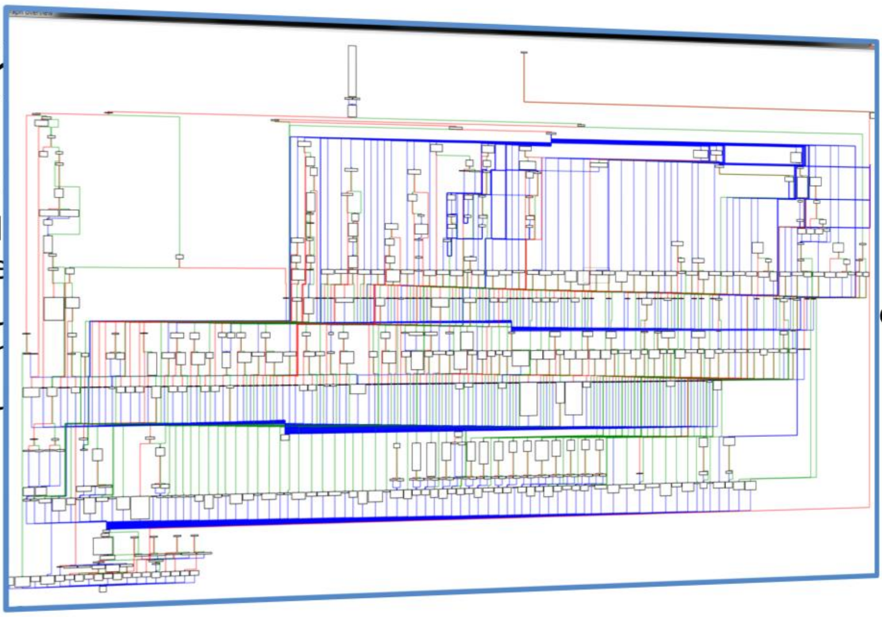
- All bugs found by fuzzing are real bugs, even if they are not security bugs
- Fuzzing does not require deep understanding of the underlying protocol or format (and often rewards ignorance)
- Fuzzing lends itself to automation

The core is, it finds bugs. From an attacker's perspective, most bugs found are useless, especially if the software was built with modern mitigations. However, the defender's dilemma means that only one useful bug is required. And while fuzzing requires enough of a knowledge of the protocol to bypass content agnostic mitigations (checksums, crcs, etc), not only does it not require a deep understanding of the protocol beyond that, it often punishes too much knowledge. If the fuzzer is too aware of how things are "supposed to be", it will miss bugs. And finally, fuzzing is a great tool to leverage on top of test automation. Obviously, for an attacker, who doesn't have test automation for a product, a fuzzing automation suite is built instead. Either way, the fuzzers can run constantly, and the bugs can be harvested for analysis when time permits.

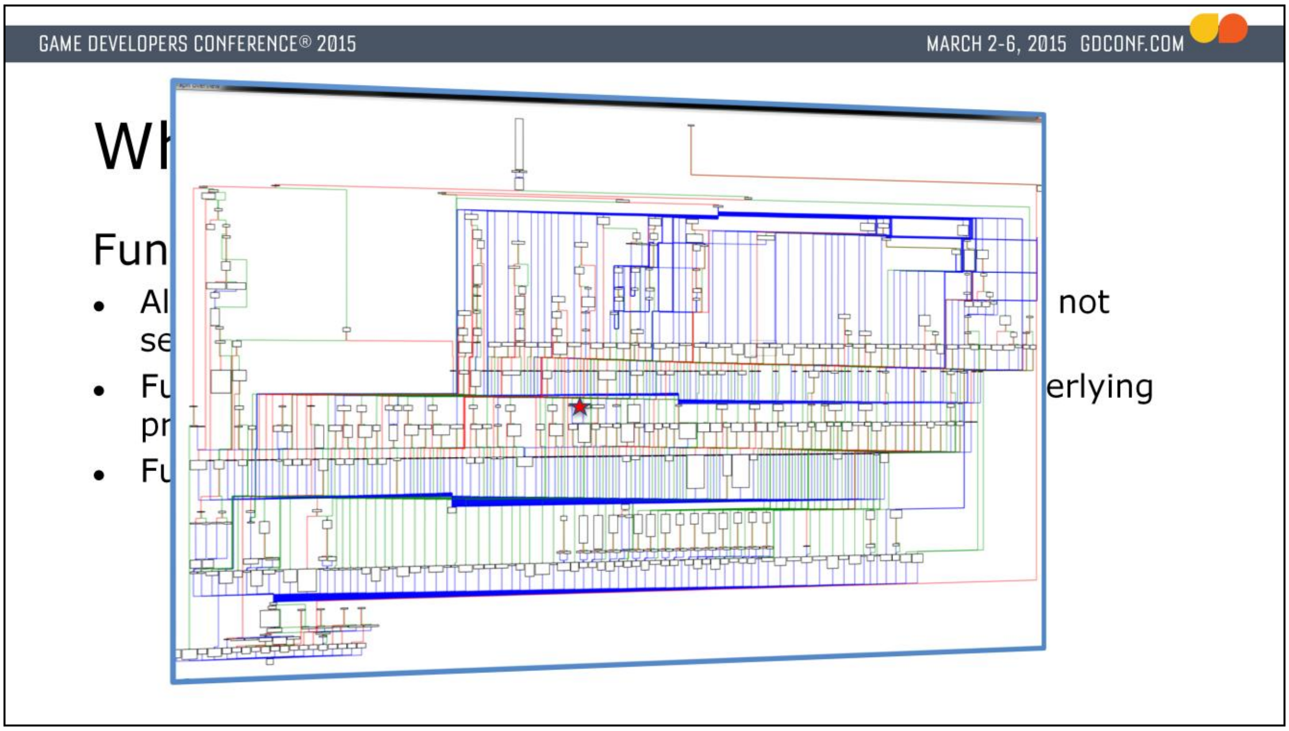
Wh

Fun

- Al not
  se
- Fu erlying
  pr
- Fu



This is the flow graph of a single parsing function, in a network facing process running as SYSTEM.

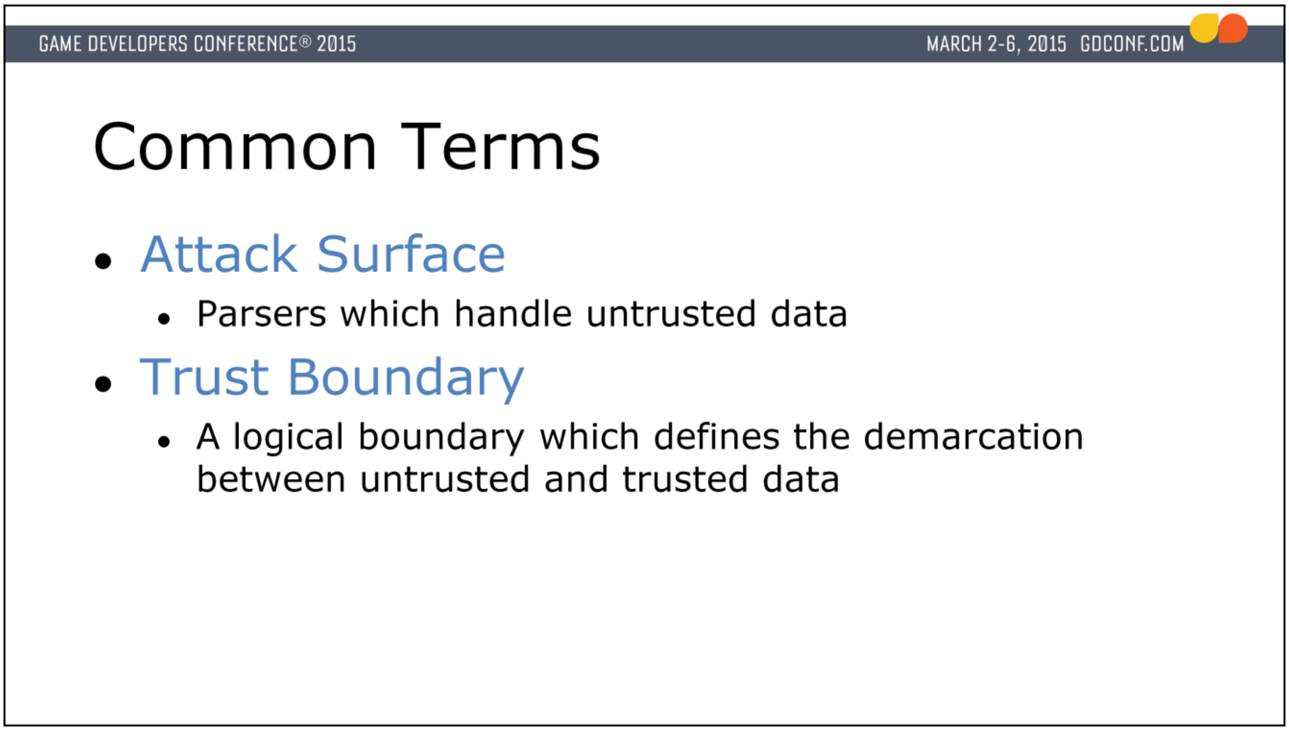


Wh

Fun

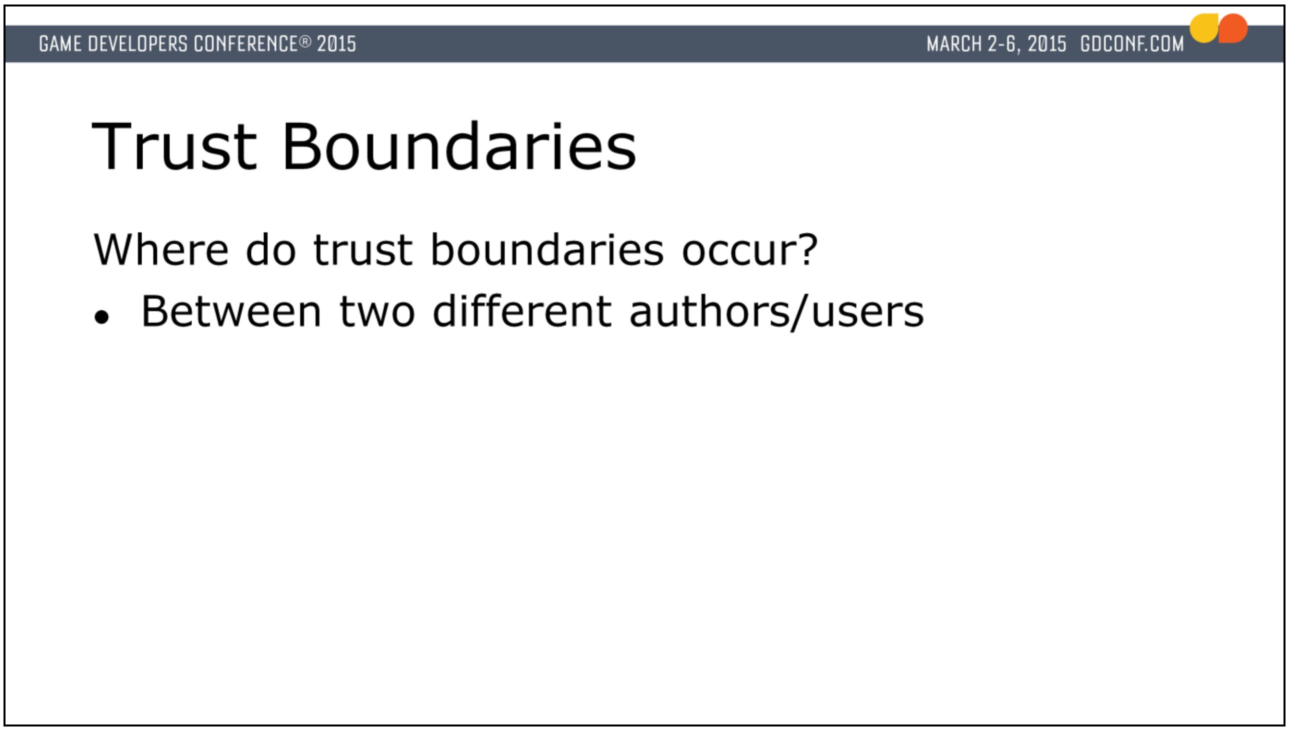
- Al not se
- Fu erlying pr
- Fu

This is the section that was vulnerable to a remote attacker getting arbitrary code execution by overflowing a static buffer. Manually audits of complex parsers are expensive, difficult, and require a difficult to acquire skill. This is the sort of thing that fuzzers excel at finding.



Common Terms

- Attack Surface
  - Parsers which handle untrusted data
- Trust Boundary
  - A logical boundary which defines the demarcation between untrusted and trusted data


GAME DEVELOPERS CONFERENCE® 2015

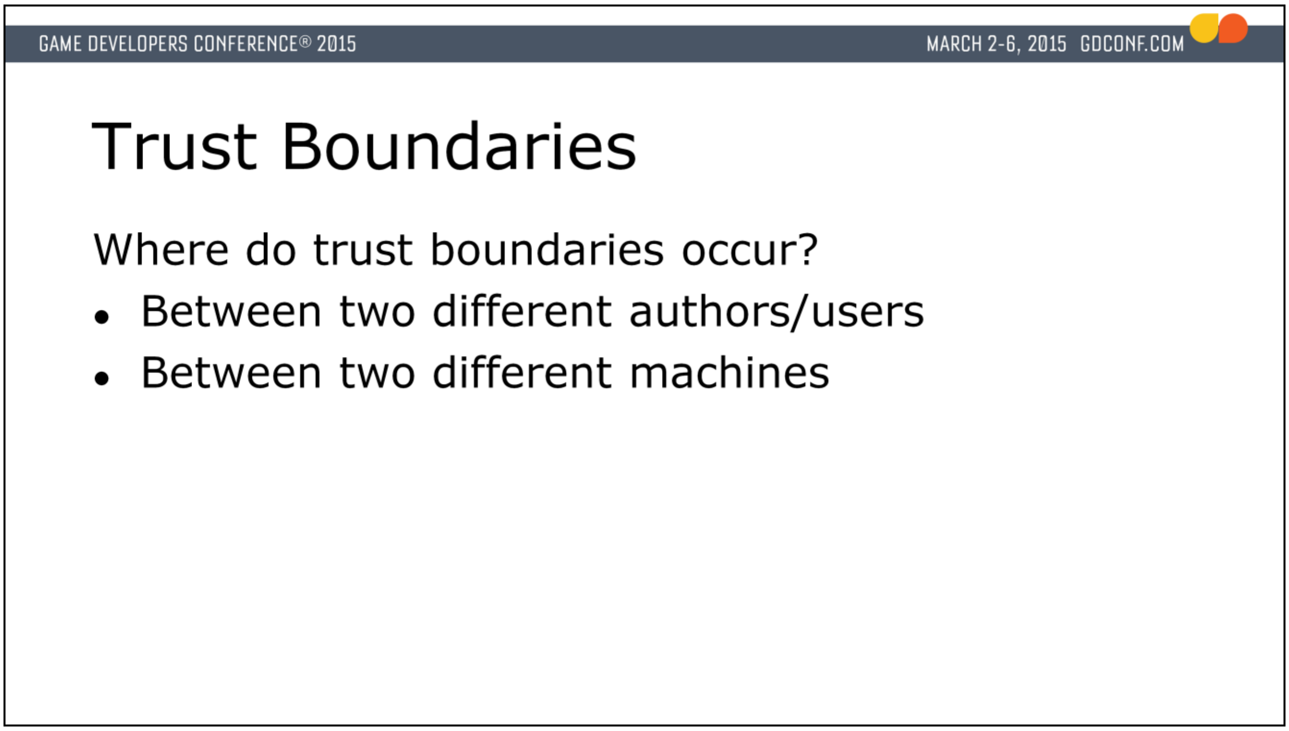
MARCH 2-6, 2015 GDCONF.COM

Trust Boundaries

Where do trust boundaries occur?

- Between two different authors/users

The simple rule of thumb is that data controls code flow. If we didn't take different actions based on data, we wouldn't use the data at all. As an attacker, I want to use the data to drive code flow in ways that you as the developer did not intend. If you are parsing data I provide ("untrusted data"), you have to assume it is hostile. If you are parsing data you provide, you don't have to, as long as you can prove it wasn't tampered with. This is why signing the developer provided data is a win, as we'll see later, you can skip fuzzing signed data (although you should fuzz the signature verification code at some point).



Trust Boundaries

Where do trust boundaries occur?

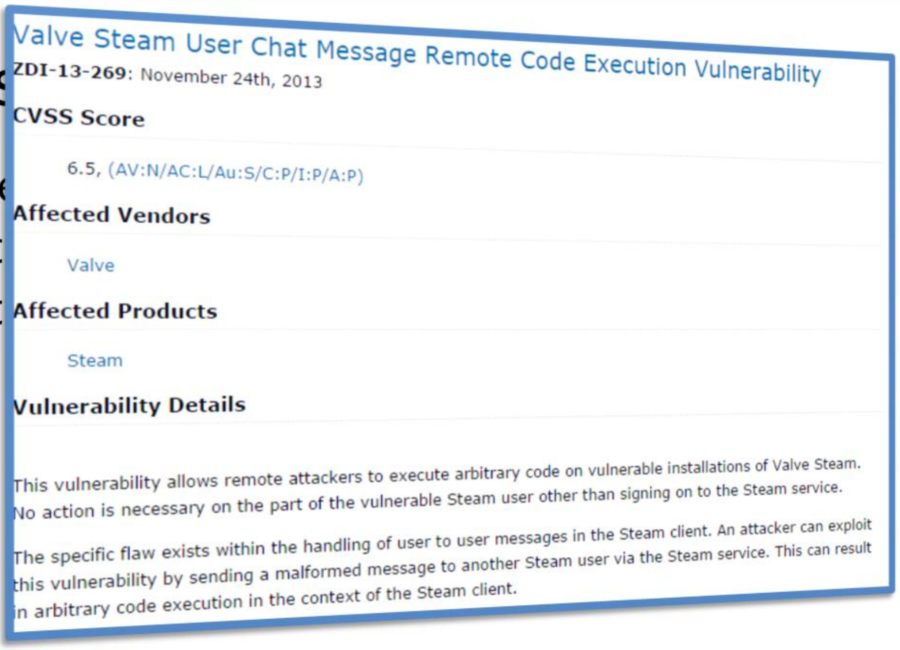
- Between two different authors/users
- Between two different machines

If the data is coming from two different machines, you need to consider it hostile (even if it is believed to be coming from the developer's server) because there are too many ways for an attacker to inject the data.

Trus

Where

- Bet
- Bet



Valve Steam User Chat Message Remote Code Execution Vulnerability

ZDI-13-269: November 24th, 2013

CVSS Score

6.5, (AV:N/AC:L/Au:S/C:P/I:P/A:P)

Affected Vendors

Valve

Affected Products

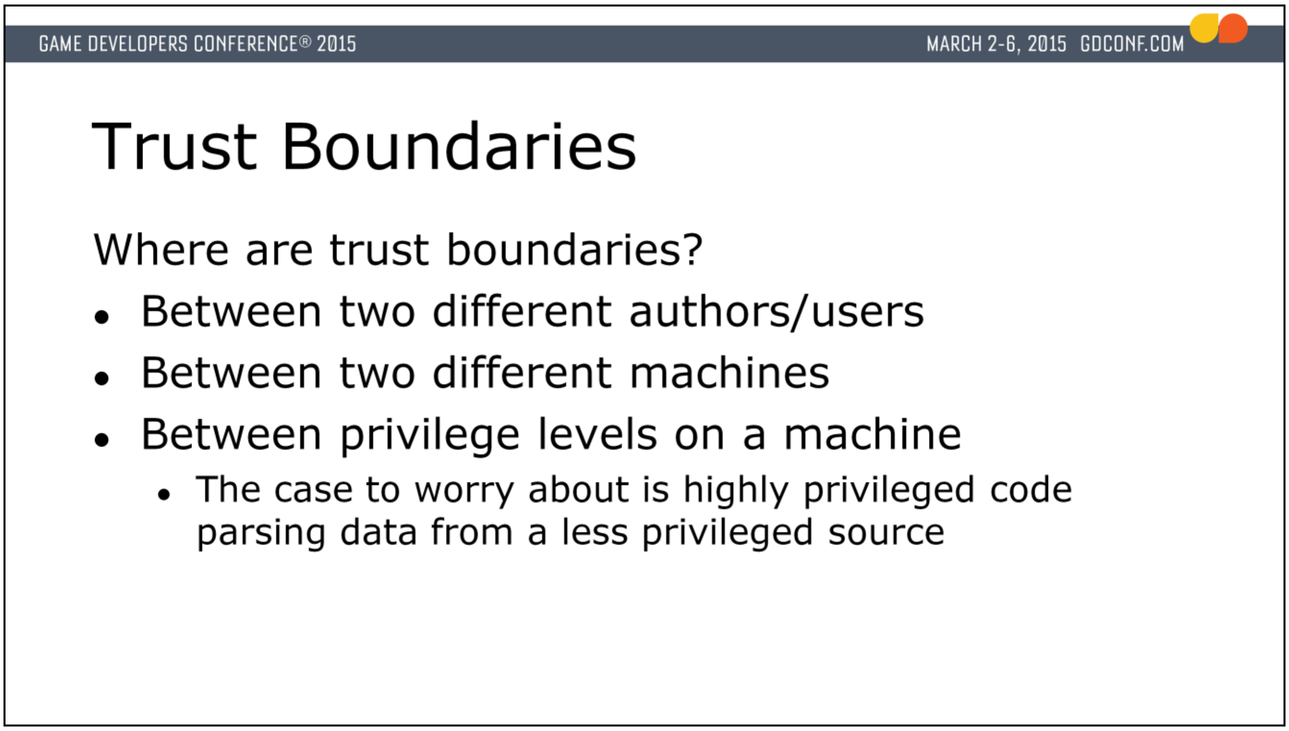
Steam

Vulnerability Details

This vulnerability allows remote attackers to execute arbitrary code on vulnerable installations of Valve Steam. No action is necessary on the part of the vulnerable Steam user other than signing on to the Steam service.

The specific flaw exists within the handling of user to user messages in the Steam client. An attacker can exploit this vulnerability by sending a malformed message to another Steam user via the Steam service. This can result in arbitrary code execution in the context of the Steam client.

This is an example of a client-server-client vulnerability. The hostile data was provided by an attacker, but forwarded by the server as a fully legitimate method. Had this been discovered and used by an attacker, rather than by a security researcher, it could have been used to create a Steam based worm.

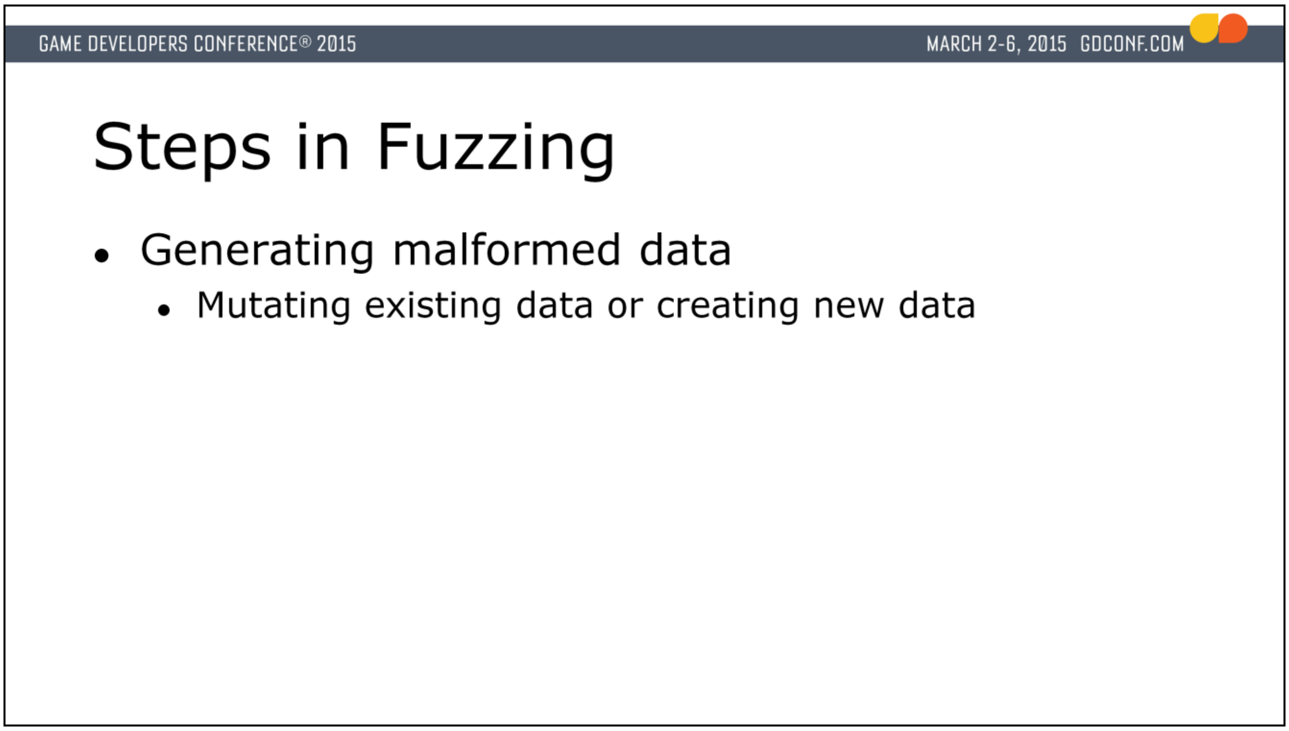


Trust Boundaries

Where are trust boundaries?

- Between two different authors/users
- Between two different machines
- Between privilege levels on a machine
  - The case to worry about is highly privileged code parsing data from a less privileged source

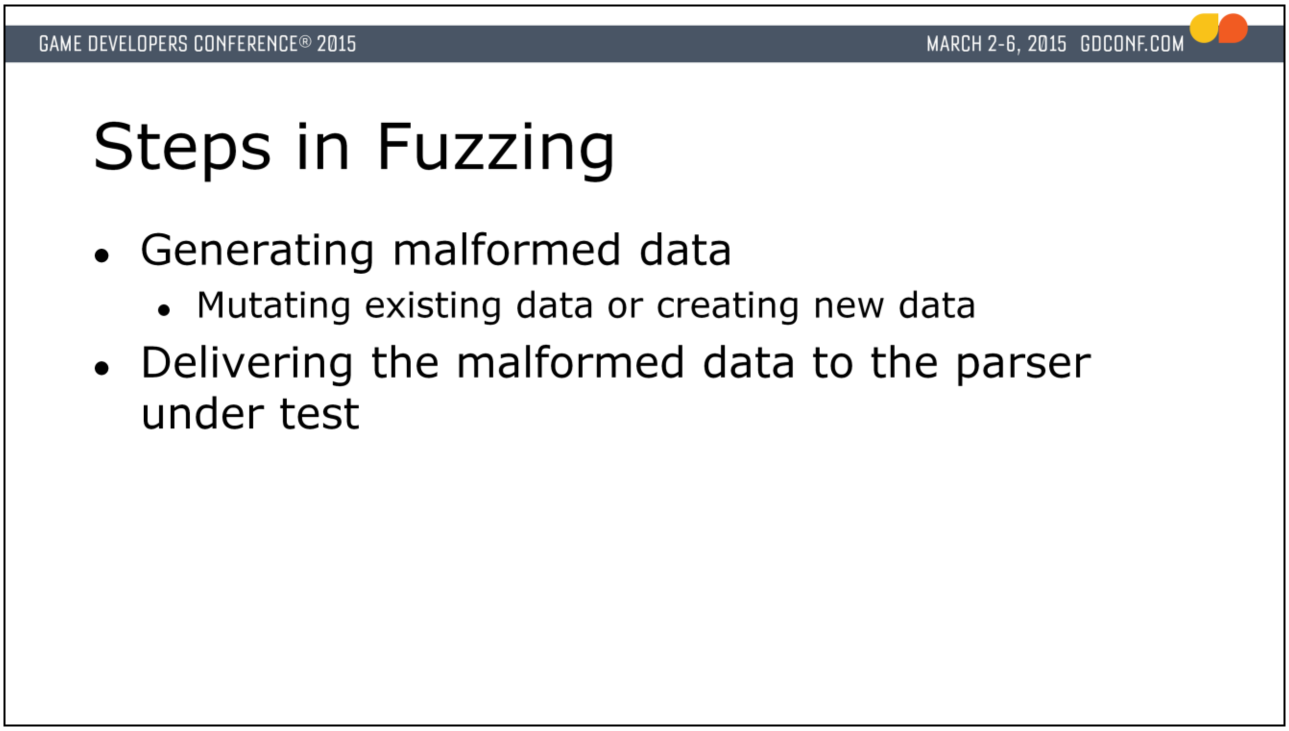
Finally, we look at privilege levels on the machine. For example, if an attacker can provide data from an unprivileged account that is used by a game component (like an updater) that runs with higher privileges, we need to worry about that. There is an exception here. We don't need to worry about files that can only be written or modified by a highly privileged account (root, Administrator, etc). If someone already has privileges on the system, they don't need to attack you. From a practical perspective, anything loaded from Program Files on Windows is safe, unless you have deliberately weakened those protections.



Steps in Fuzzing

- Generating malformed data
  - Mutating existing data or creating new data

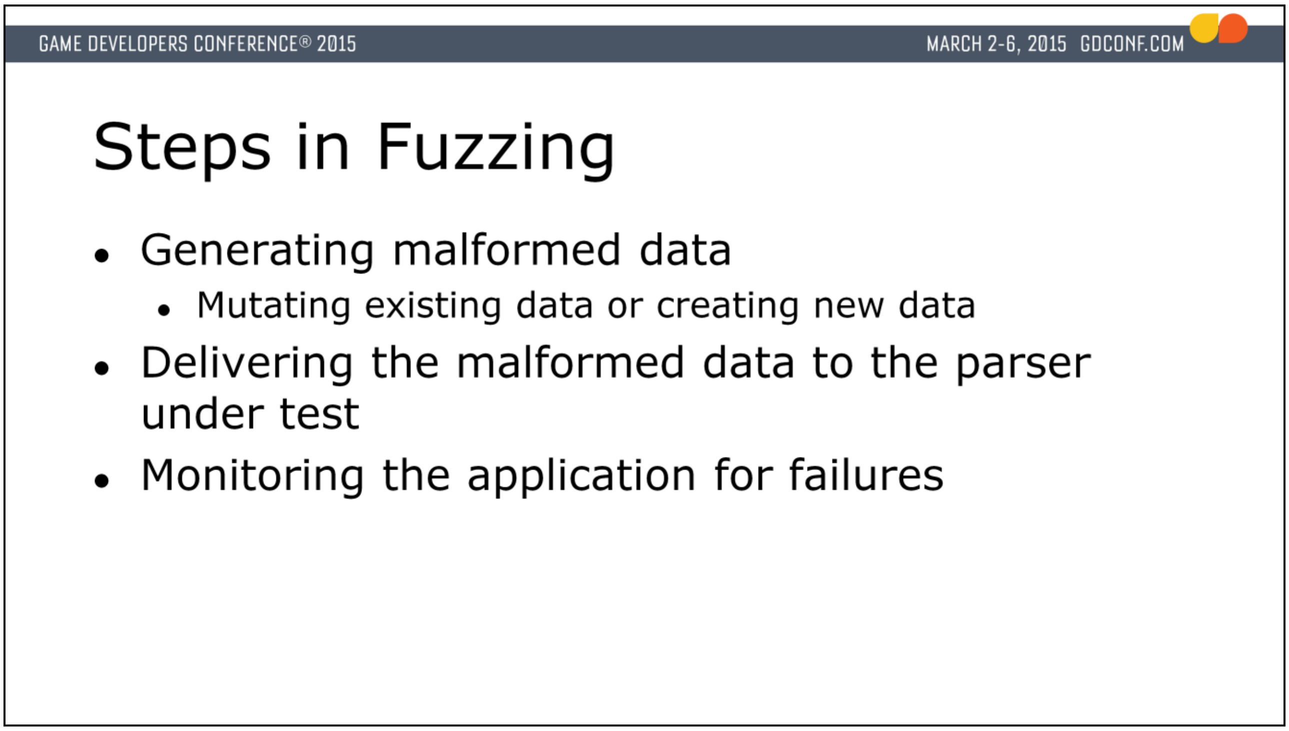
Whether mutating existing data is easier or creating new data is easier depends on what you have available to you. If you have an existing wide range of test data (or a corpus of real data, like, say, all the content for the game), mutation is easier. This is why attackers often use generation fuzzing for network fuzzing (which usually requires state tracking), but mutation fuzzing for files (which doesn't, and often has examples)



Steps in Fuzzing

- Generating malformed data
  - Mutating existing data or creating new data
- Delivering the malformed data to the parser under test

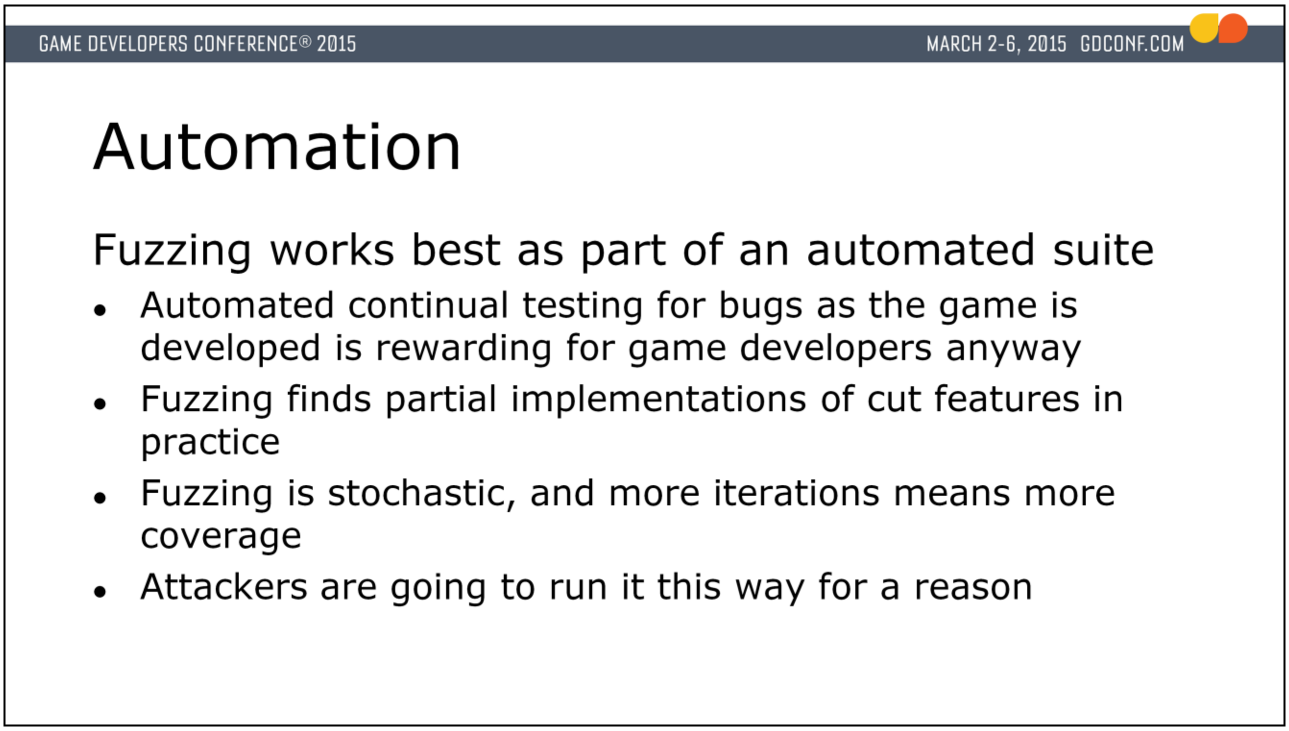
The goal is to get data to the parser we care about, and get it there quickly. This means it can make sense to build testbeds that are just the parser you care about, or to have build versions or command line arguments that let you pass data straight to the parser you need. You'll want to run the parser quickly, and then terminate the process after a short period of time (or have it parse the data and then exit normally), so that you can run as many iterations as possible.


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Steps in Fuzzing

- Generating malformed data
  - Mutating existing data or creating new data
- Delivering the malformed data to the parser under test
- Monitoring the application for failures

Finally, you need to know when something bad happens. The obvious case is an actual crash, but you also want to look at things like memory usage (if I can send a network packet that causes you to allocate 2 gigabytes of RAM, that's an issue) and CPU usage. In all of these cases, you want to monitor the process under test. When crashes happen, you need to save as much data as you can about the crash so that you can reproduce it. In the case of files, the obvious thing to save is the malformed file. For network fuzzing, since it has a fair amount of entropy in it, you'll want to save things like the random number seed, the settings, and as much as possible about the changes that were made.

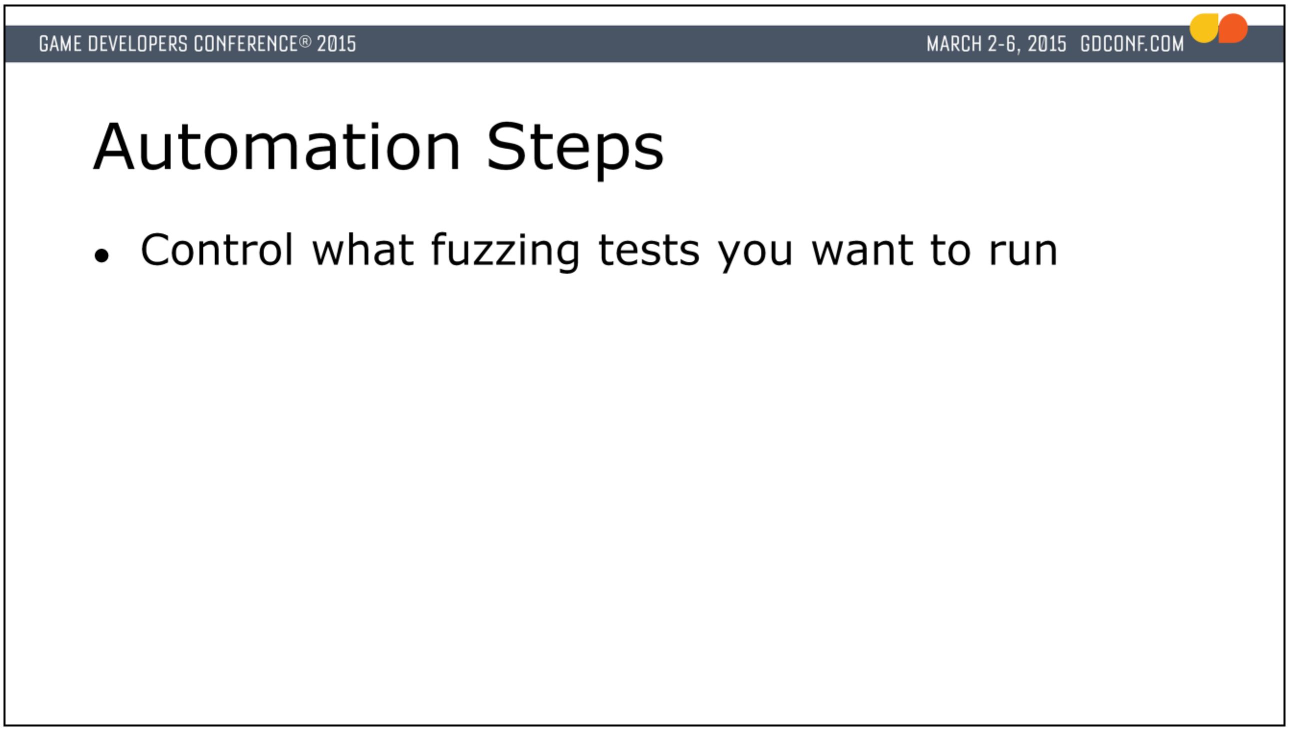

GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Automation

Fuzzing works best as part of an automated suite

- Automated continual testing for bugs as the game is developed is rewarding for game developers anyway
- Fuzzing finds partial implementations of cut features in practice
- Fuzzing is stochastic, and more iterations means more coverage
- Attackers are going to run it this way for a reason

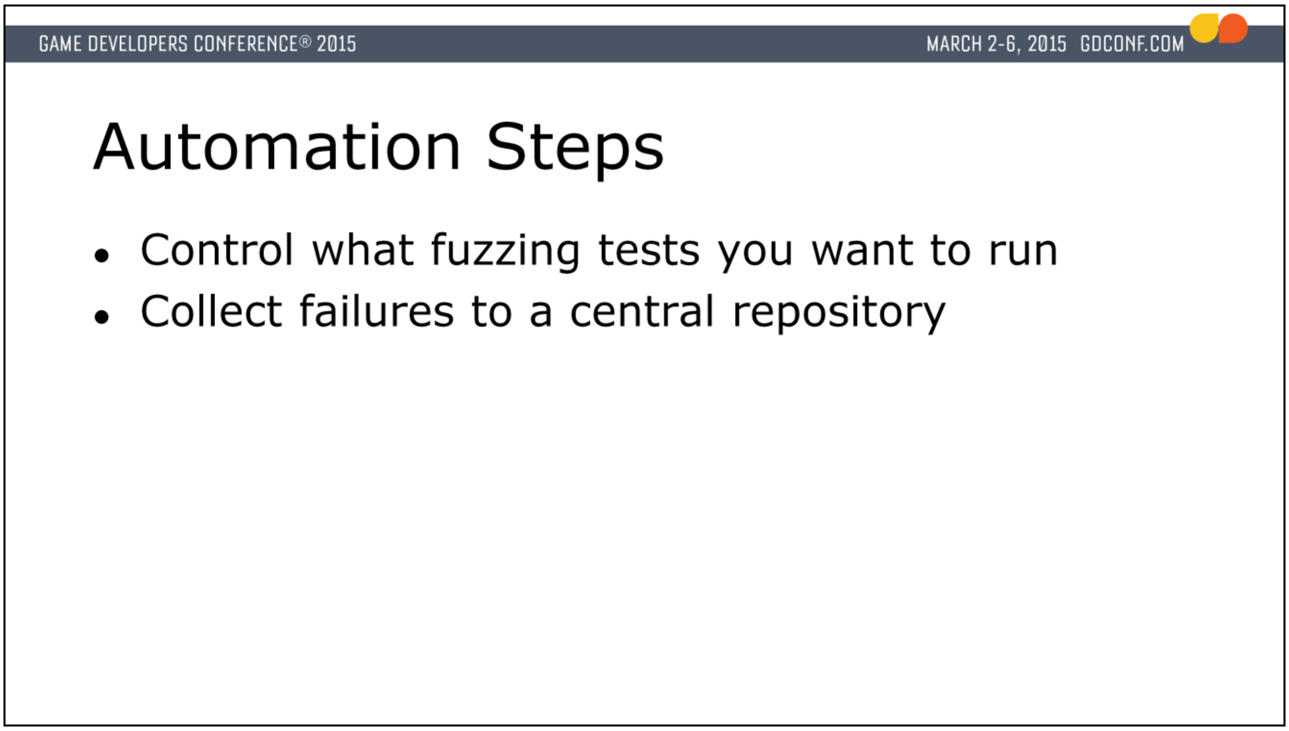
While ad-hoc fuzzing (especially for network traffic) is possible, the biggest return on your investment is to invest in an automation infrastructure. For file fuzzing, this is relatively easy. For network fuzzing, you have more work to do. If your multiplayer game supports AI players, it can be a big help if the AIs can run on different computers, and play against each other.


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Automation Steps

- Control what fuzzing tests you want to run

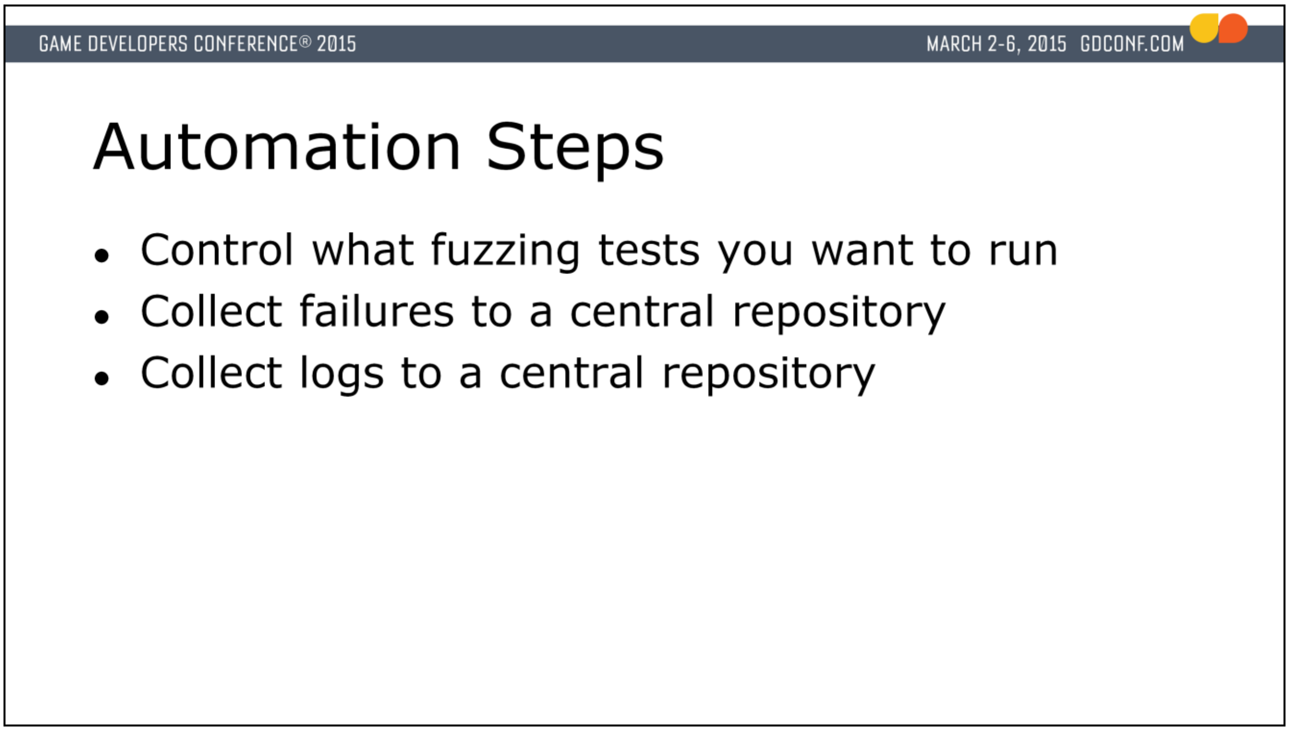
You will likely have a whole set of different parsers to fuzz. Each will likely have a different set of configuration information (a corpus of test files, or a set of settings for the command line, and so on). You want to have a central place to manage this from, even if it is just a web page of config files the machines can pull from



Automation Steps

- Control what fuzzing tests you want to run
- Collect failures to a central repository

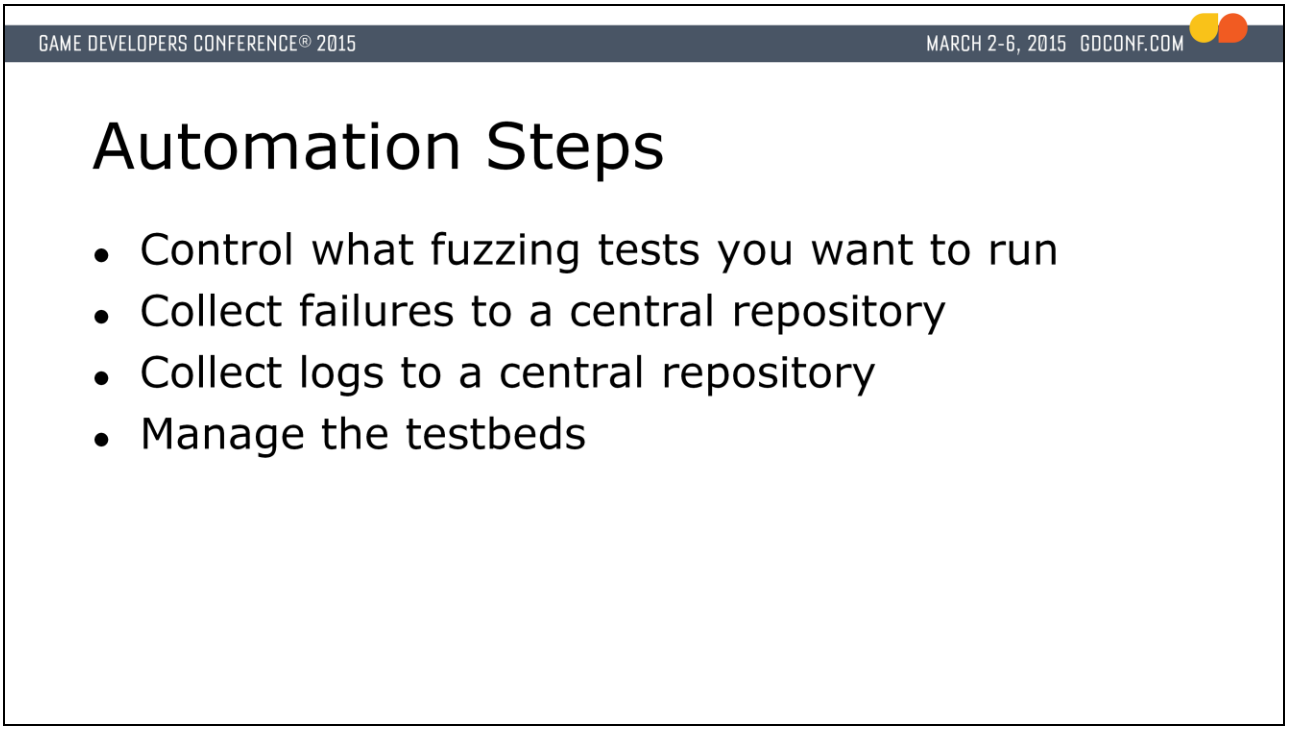
Put all your crashes and other related information in one central place. This lets you have machines join your fuzzing farm part time (for example, when the devs go get some actual sleep), but not affect a machine when it is being used. Plus, it lets you do crash triage with one set of crashes.



Automation Steps

- Control what fuzzing tests you want to run
- Collect failures to a central repository
- Collect logs to a central repository

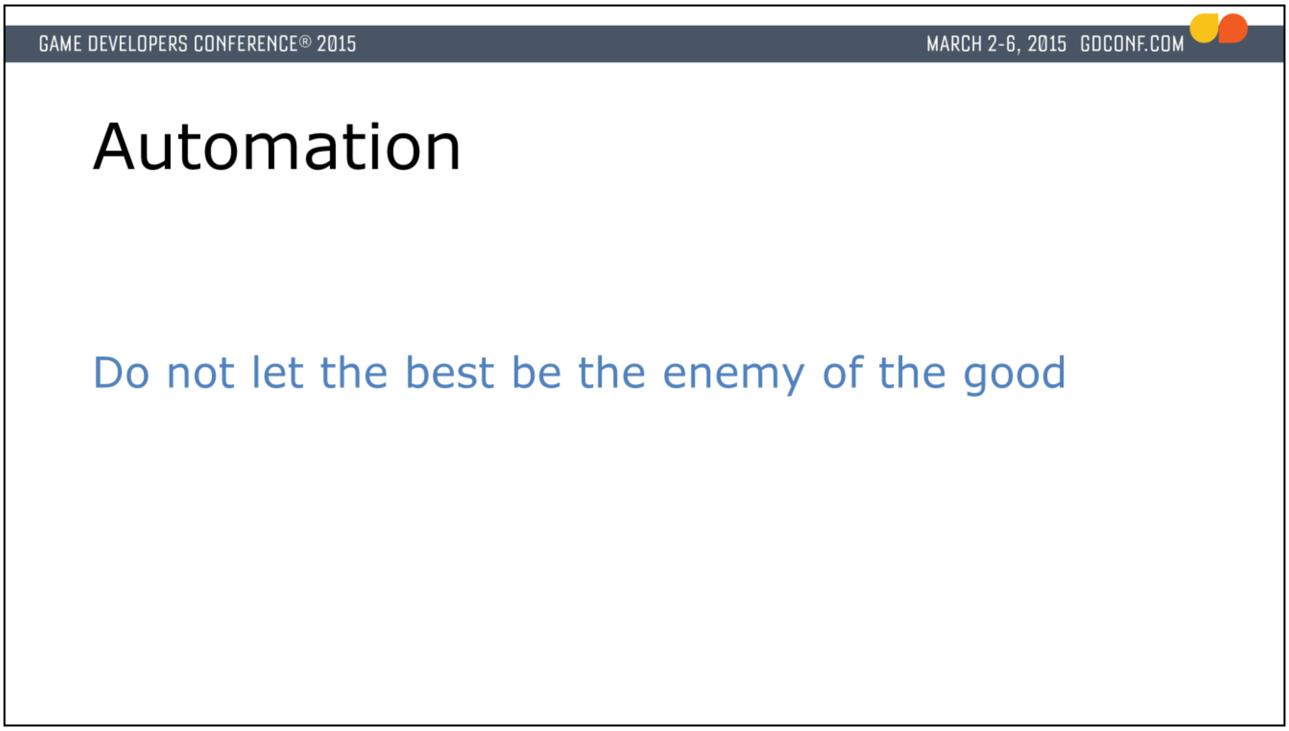
For the same reason, save the logs (and correlate them with crashes). Ideally, you should also keep stats on what configurations aren't finding bugs, and review those.



Automation Steps

- Control what fuzzing tests you want to run
- Collect failures to a central repository
- Collect logs to a central repository
- Manage the testbeds

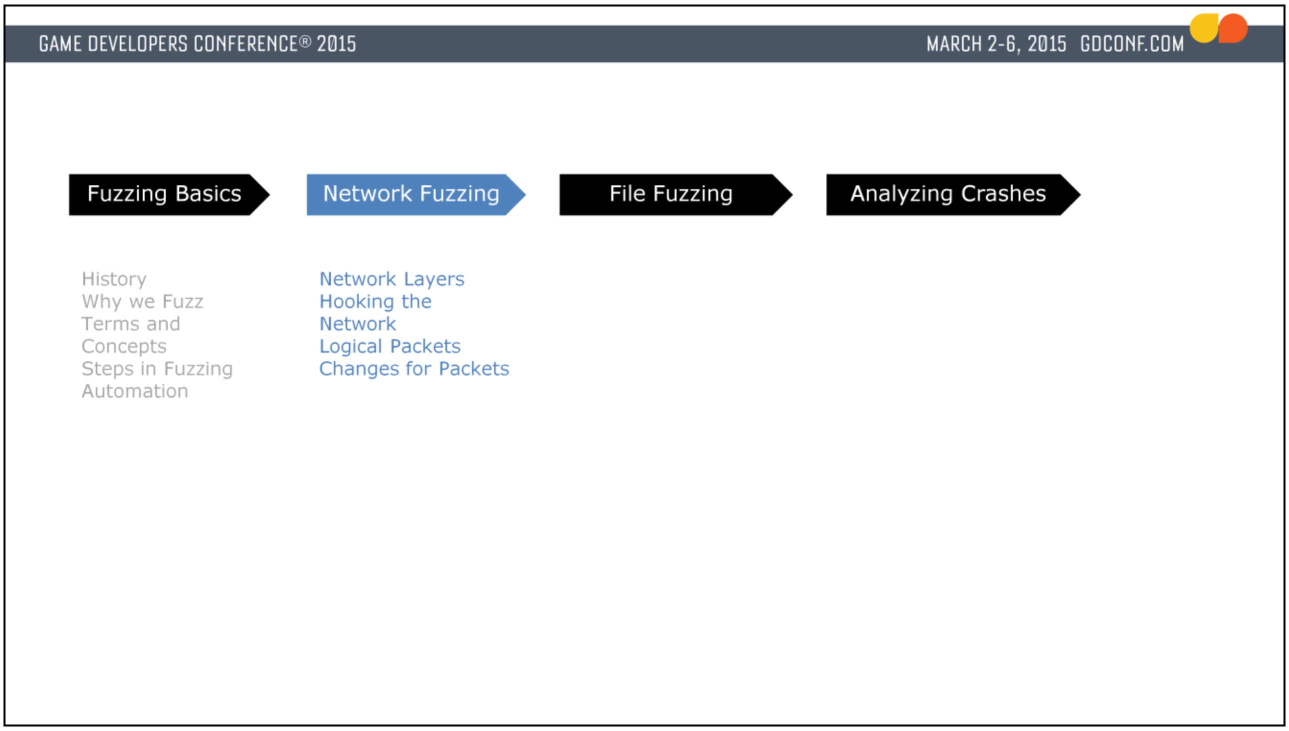
At the very top of the "wish list" for automation is single point control of your fuzzing. This can be done these days with a simple web page and web controls, and you likely already have the expertise to implement this kind of web based light client in house already. This would let you have a config-of-configs, and fuzz farm elements would use that to determine what sort of fuzzing they should do, and then get the right configuration.



Automation

Do not let the best be the enemy of the good

The automation steps I just outlined are an ideal. They may well be too much of an investment up front. In the simplest case, you want to simply run tests on one machine (or two, if you are testing network code), with tests running under a simple test harness that simply runs the iterations.



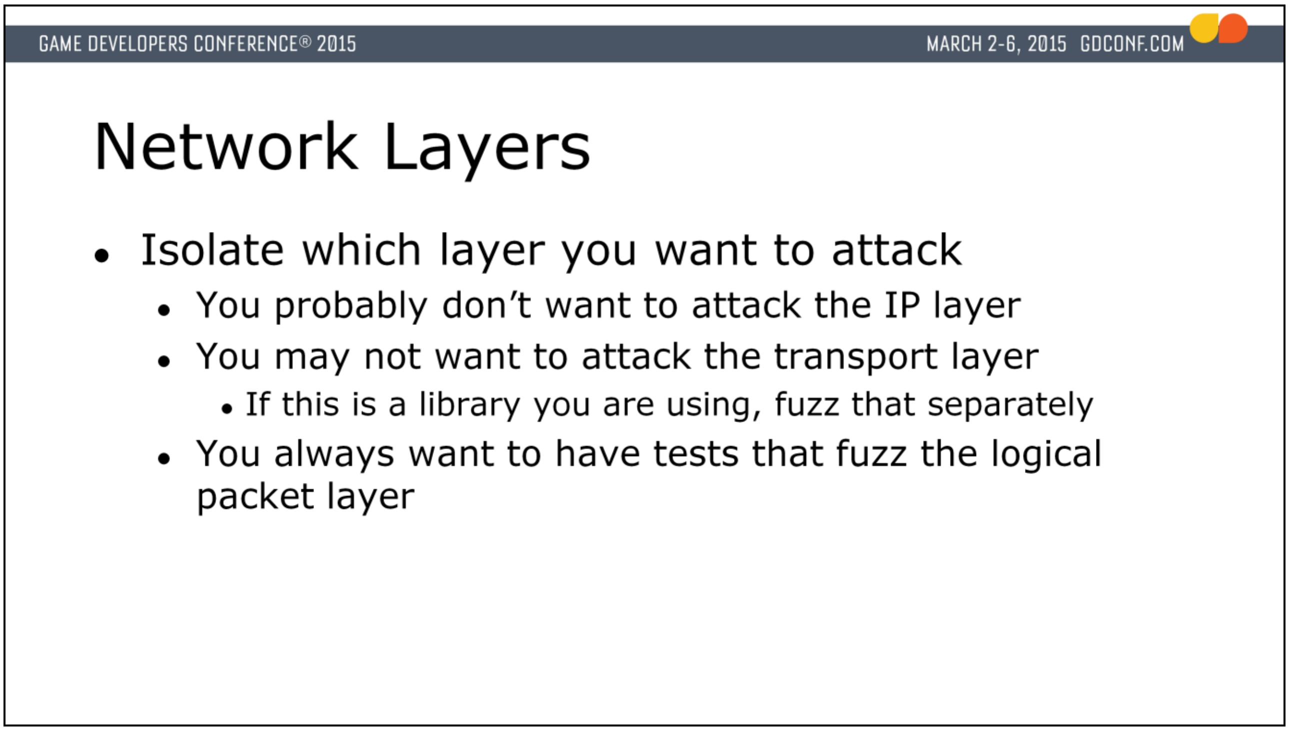
GAME DEVELOPERS CONFERENCE® 2015

MARCH 2-6, 2015 GDCONF.COM

Fuzzing Basics | Network Fuzzing | File Fuzzing | Analyzing Crashes

History
Why we Fuzz
Terms and Concepts
Steps in Fuzzing
Automation

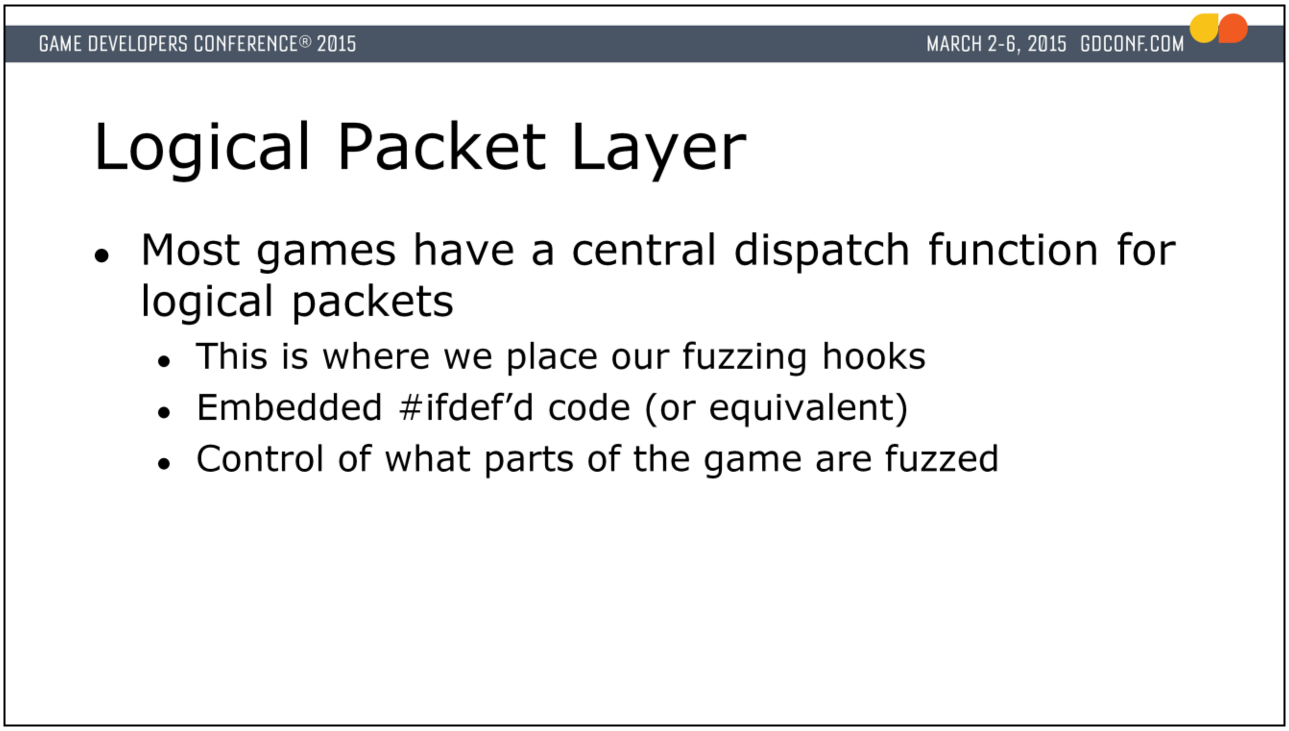
Network Layers
Hooking the Network
Logical Packets
Changes for Packets


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Network Layers

- Isolate which layer you want to attack
  - You probably don’t want to attack the IP layer
  - You may not want to attack the transport layer
    - If this is a library you are using, fuzz that separately
  - You always want to have tests that fuzz the logical packet layer

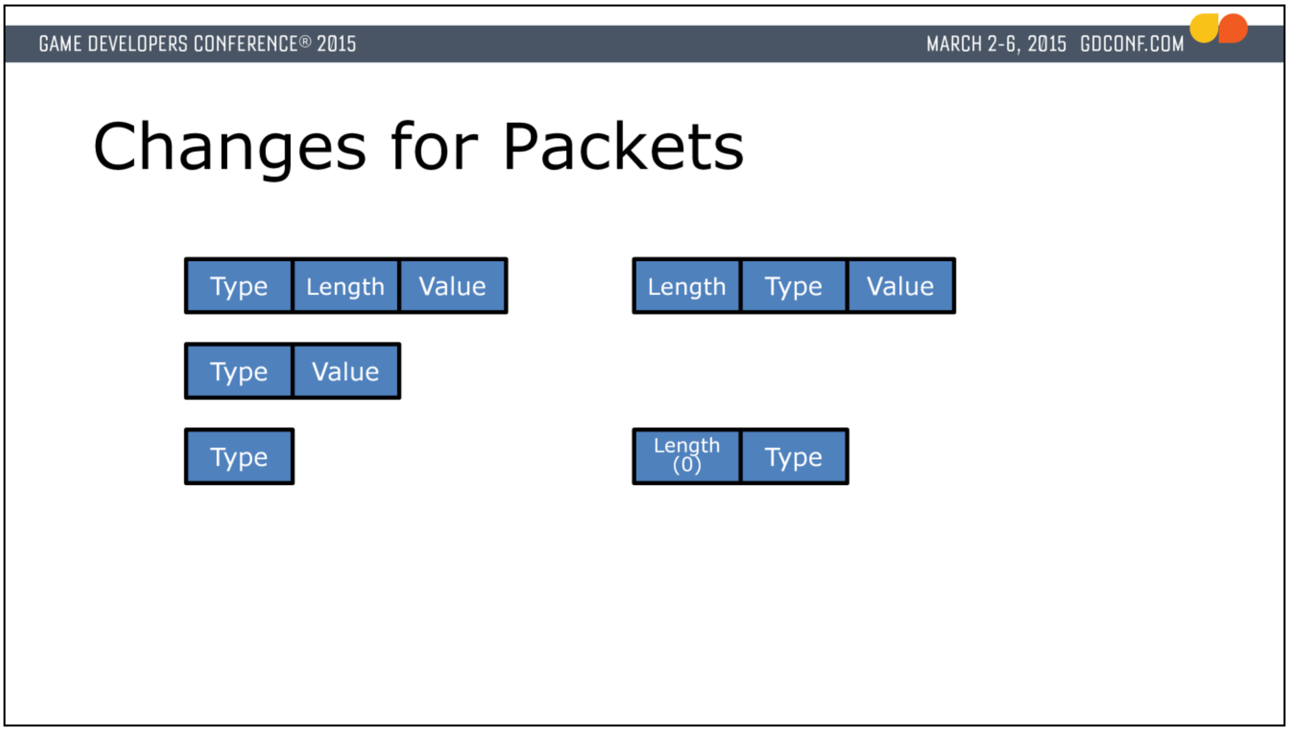
When I say that you always want to fuzz the logical layer, that doesn’t mean you want to target the logical layer when you are fuzzing the transport layer.


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Logical Packet Layer

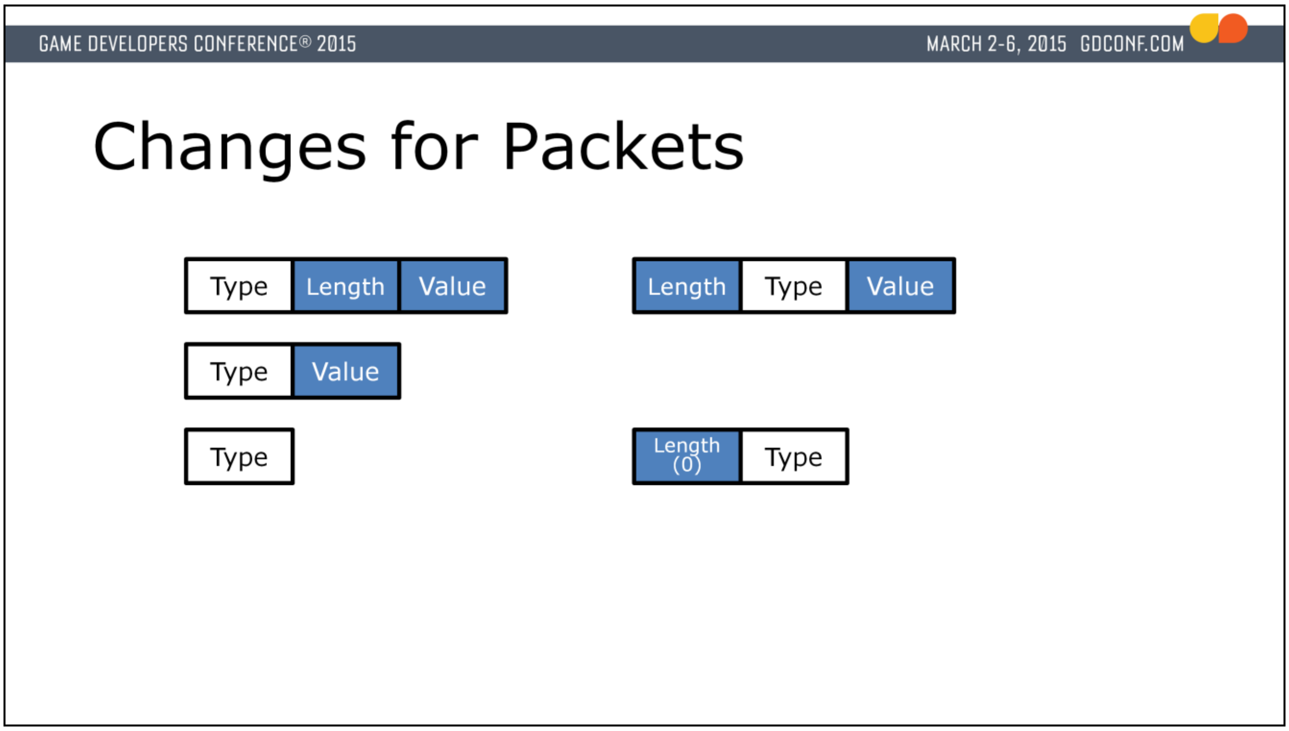
- Most games have a central dispatch function for logical packets
  - This is where we place our fuzzing hooks
  - Embedded #ifdef'd code (or equivalent)
  - Control of what parts of the game are fuzzed

Because you have the source code, it is easiest for you to do network fuzzing by doing it from inside the game. This is where you'd hook the code. You will probably want a set of options set from a control line (or at build time, if that is not viable for your platform). For example, you may want to fuzz everything, only fuzz in a lobby, only fuzz in a given game mode, or only fuzz the login/authentication mechanism.



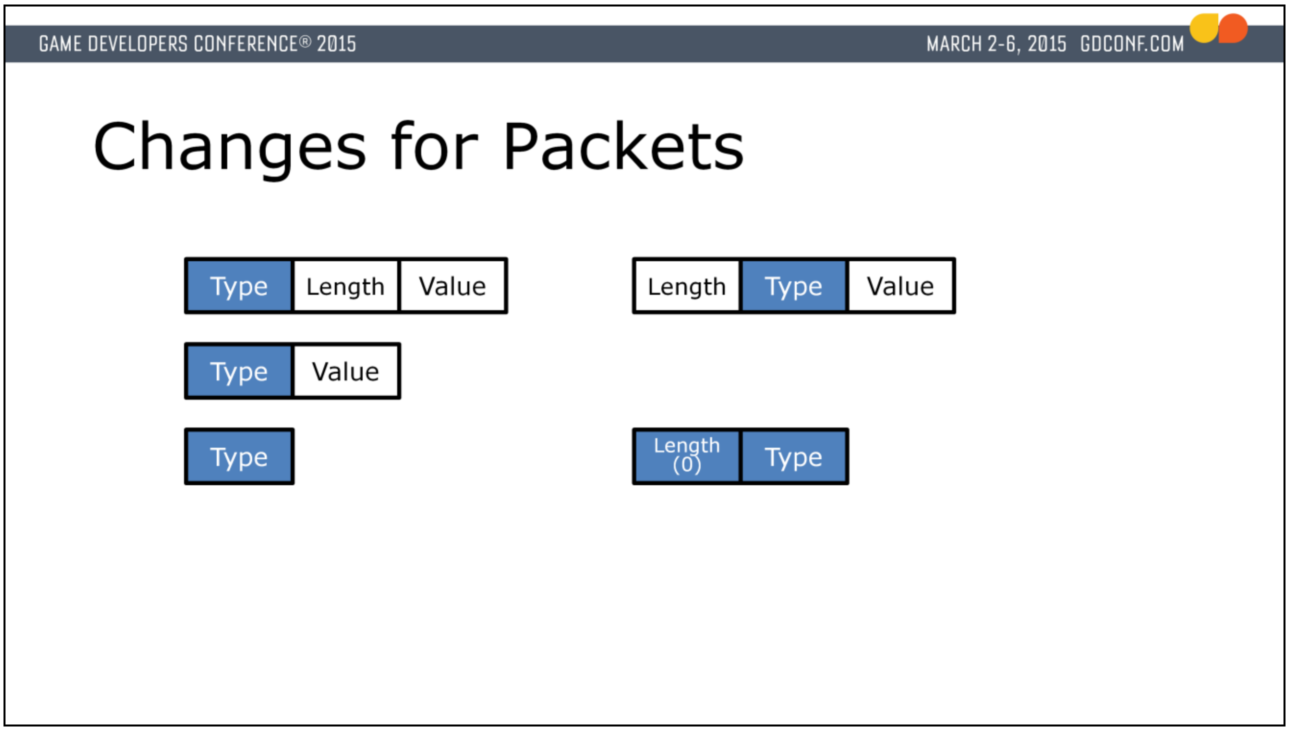
Changes for Packets

Most logical packets are in some variant of a Type/Length/Value scheme. One obvious optimization is to eliminate the length if the length is constant for a given type, and so we have a set of possible encoding schemes.


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

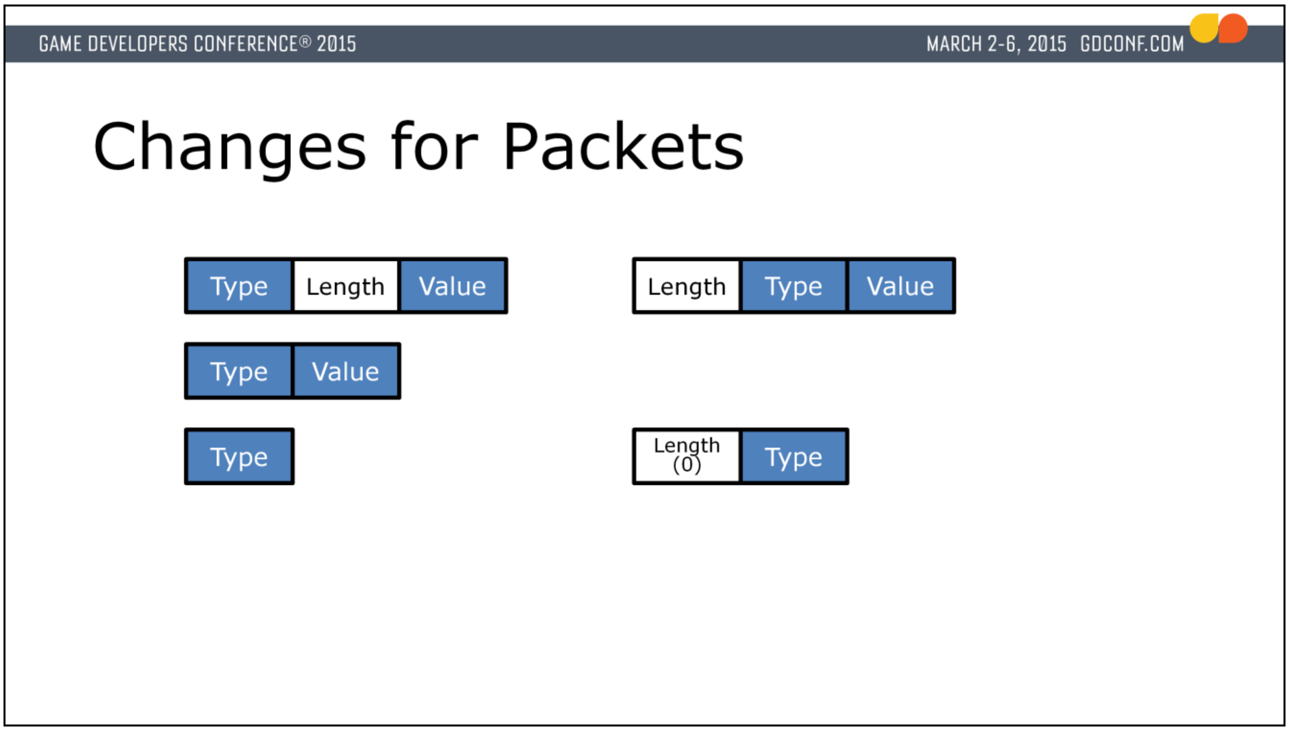
Changes for Packets

The first and easiest to implement is to randomly change the message type (the message ID) to another valid type some small percentage of the time. Leave everything else the same



Changes for Packets

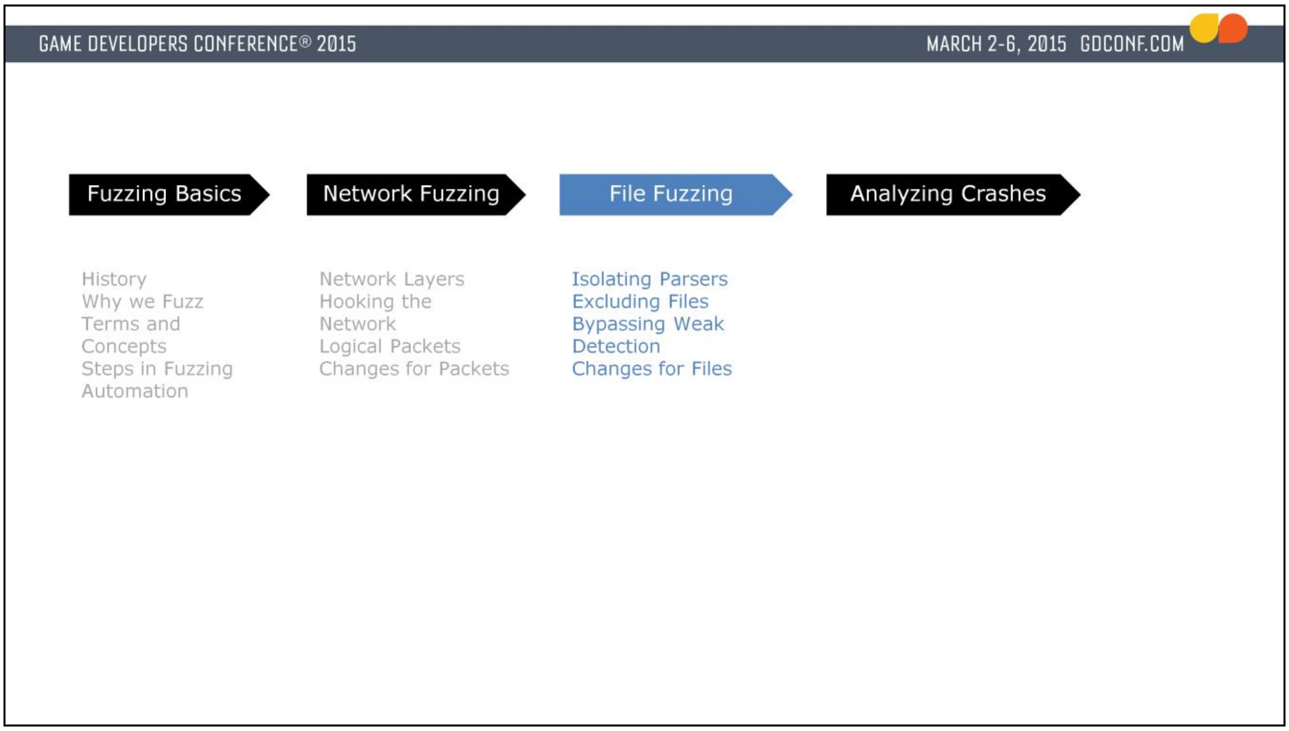
We can also malform the value of the packet. Changes can include random bit flips, replacing individual bytes (0x0, 0xff, 0xfe, 0x7f, and 0x80 are all good, as are changing a byte by one in either direction), deleting data, or injecting new data. If you change the length of the data here, go ahead and make the length field also conform to the changed data.



GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Changes for Packets

Finally, we can leave everything else alone, but just change the length field, and see what happens.



Fuzzing Basics

- History
- Why we Fuzz
- Terms and Concepts
- Steps in Fuzzing
- Automation

Network Fuzzing

- Network Layers
- Hooking the Network
- Logical Packets
- Changes for Packets

File Fuzzing

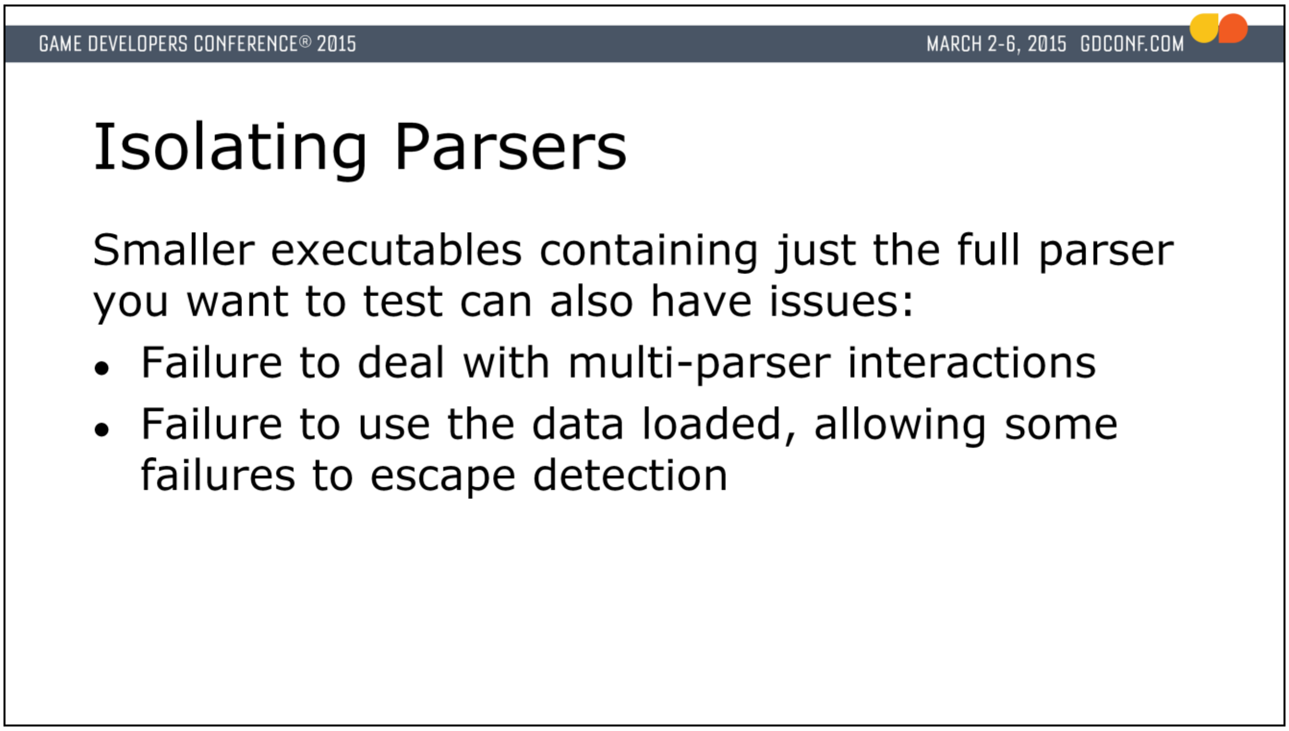
- Isolating Parsers
- Excluding Files
- Bypassing Weak Detection
- Changes for Files

Analyzing Crashes

Isolating Parsers

Smaller executables containing just the full parser you want to test (including all full allocations of data) can be useful:

- Faster test iteration
- Cleaner analysis of failures

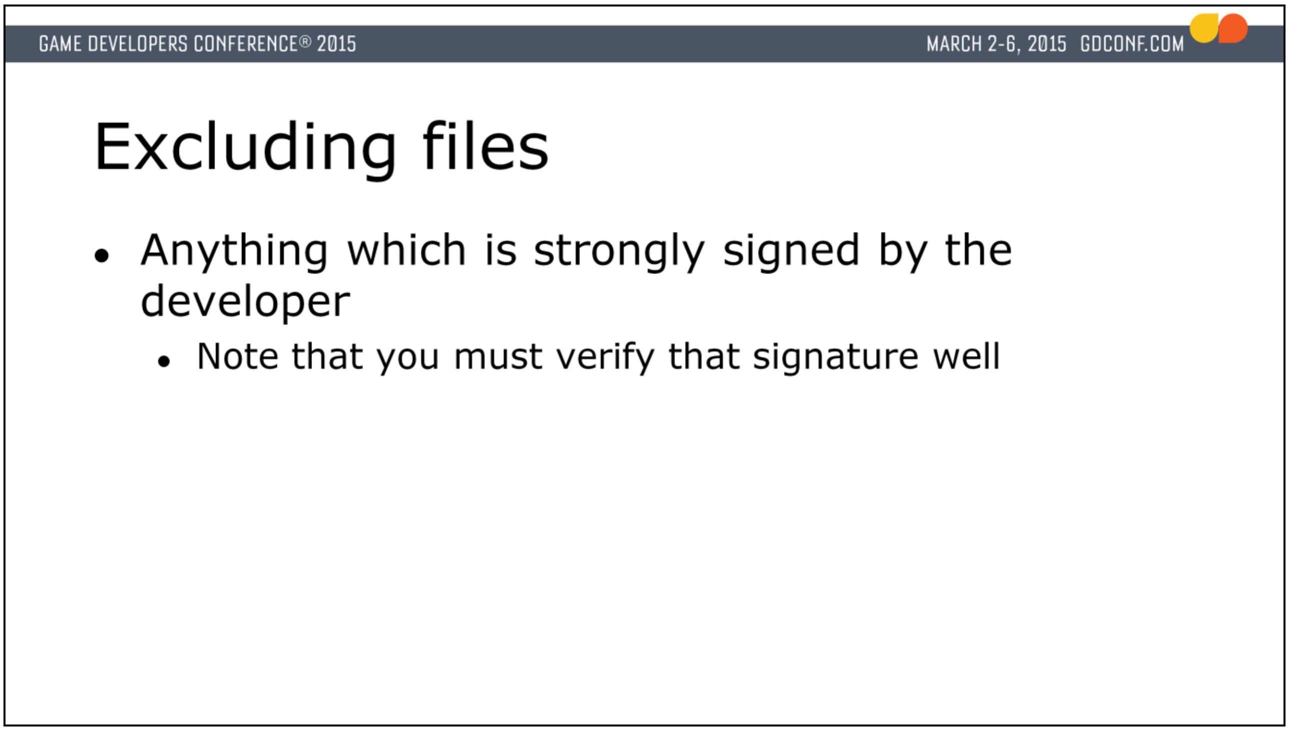


Isolating Parsers

Smaller executables containing just the full parser you want to test can also have issues:

- Failure to deal with multi-parser interactions
- Failure to use the data loaded, allowing some failures to escape detection

The obvious failure case here are “globbed” files, where an entire level is turned into a pre-combined binary blob, with pointer fixups on load. This is an incredibly dangerous structure, and absolutely must be fuzzed. Fuzzing the sub-parsers won’t get you that final fixup. The conclusion is that isolating parsers is a “nice to have” optimization for more iteration coverage, but not a replacement for testing against the full product.

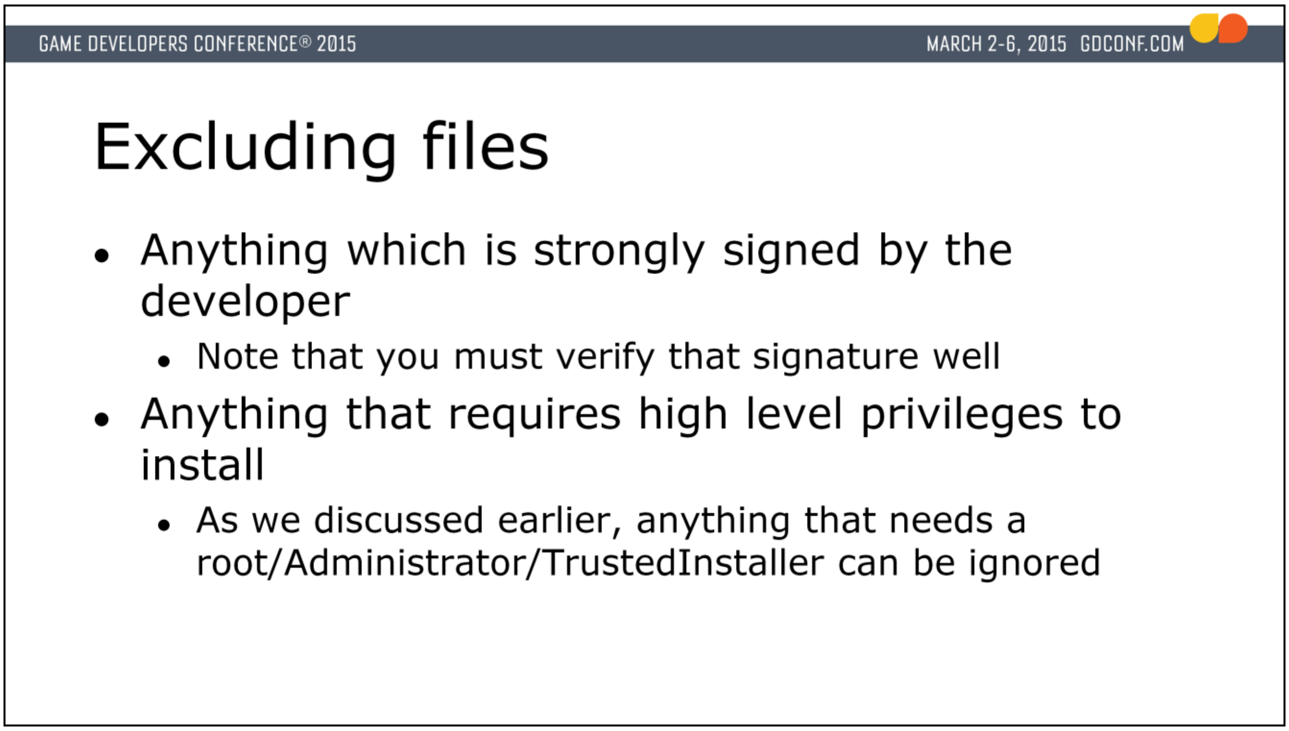


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Excluding files

- Anything which is strongly signed by the developer
  - Note that you must verify that signature well

I was once asked, by a major publisher, “do we need to verify that we signed the executable, or is it enough that it is signed”.



Excluding files

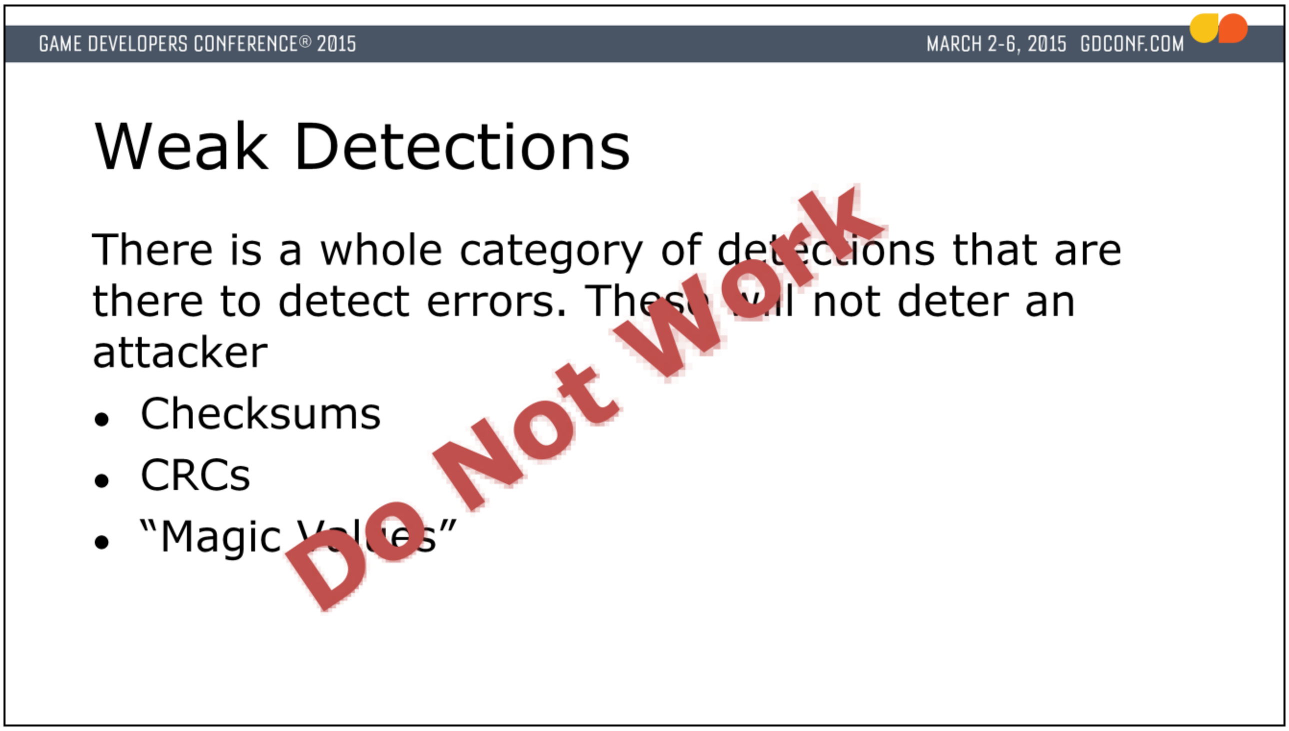
- Anything which is strongly signed by the developer
  - Note that you must verify that signature well
- Anything that requires high level privileges to install
  - As we discussed earlier, anything that needs a root/Administrator/TrustedInstaller can be ignored

We just fundamentally don't care about "root to root" exploits. If an attacker has that level of privileges, the system is already completely compromised.

Weak Detections

There is a whole category of detections that are there to detect errors. These will not deter an attacker

- Checksums
- CRCs
- “Magic Values”


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

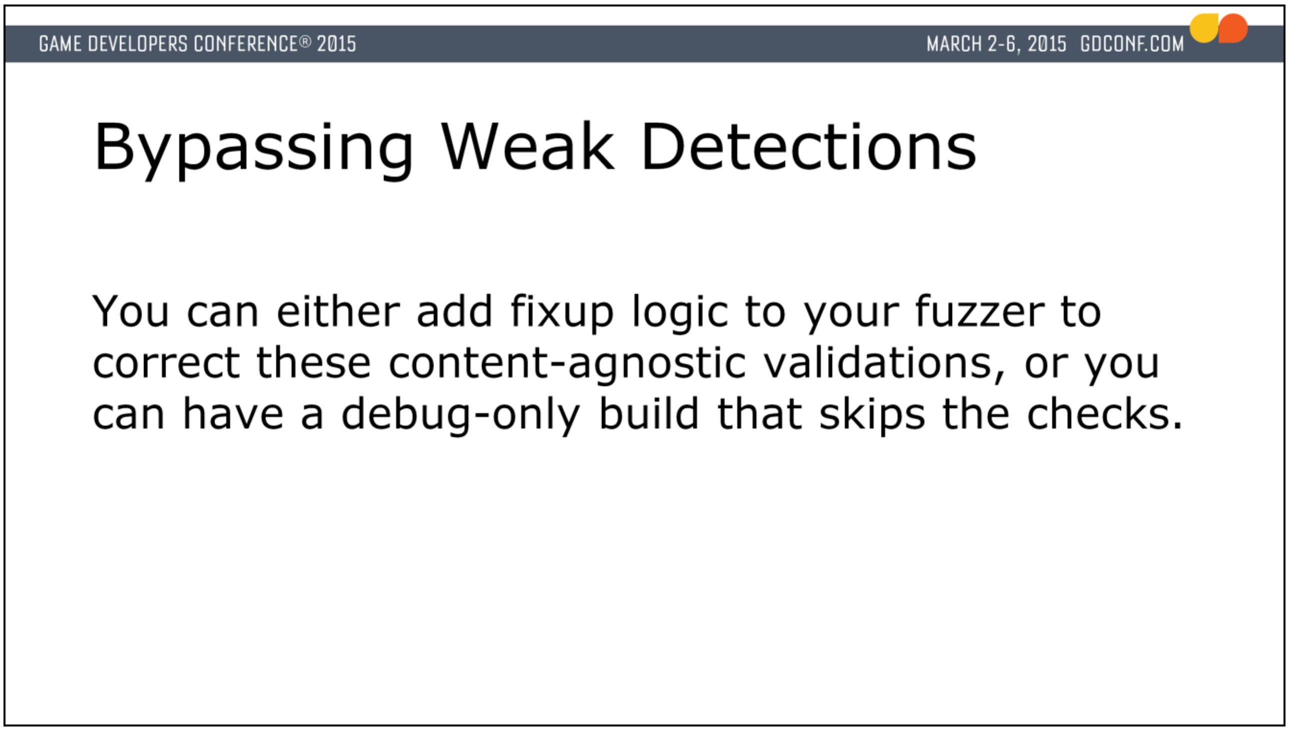
Weak Detections

There is a whole category of detections that are there to detect errors. These will not deter an attacker

- Checksums
- CRCs
- "Magic Values"

Do Not Work

Pretty much anything short of "signed by a private key that we have and you don't" isn't going to stop an attacker. Your issue isn't data corruption, it's an actual attacker. They will reverse engineer your custom CRC, and skate on by.

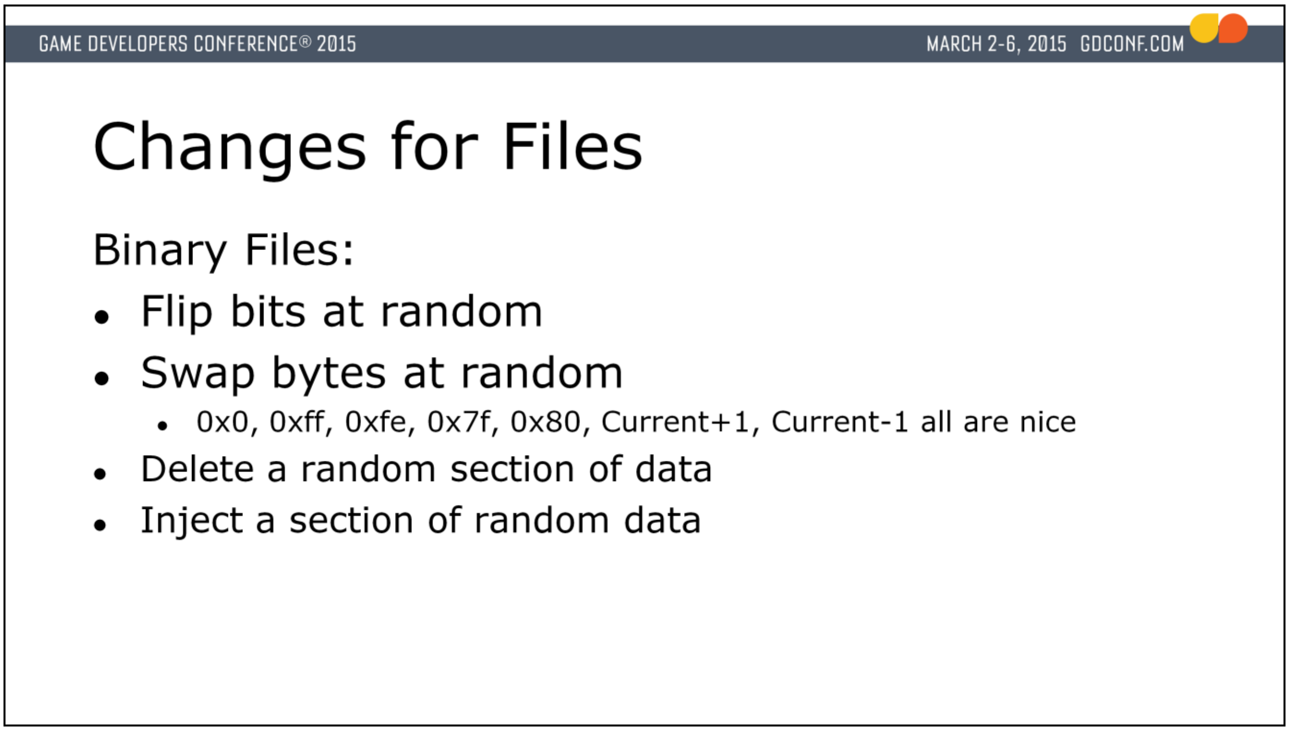


GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Bypassing Weak Detections

You can either add fixup logic to your fuzzer to correct these content-agnostic validations, or you can have a debug-only build that skips the checks.

Either of these options is good. One is significantly easier to implement. The key is that weak detections only protect you against naïve fuzzing, they do not protect you against an attacker or get you any benefit. Don't make the mistake of thinking they will save you.

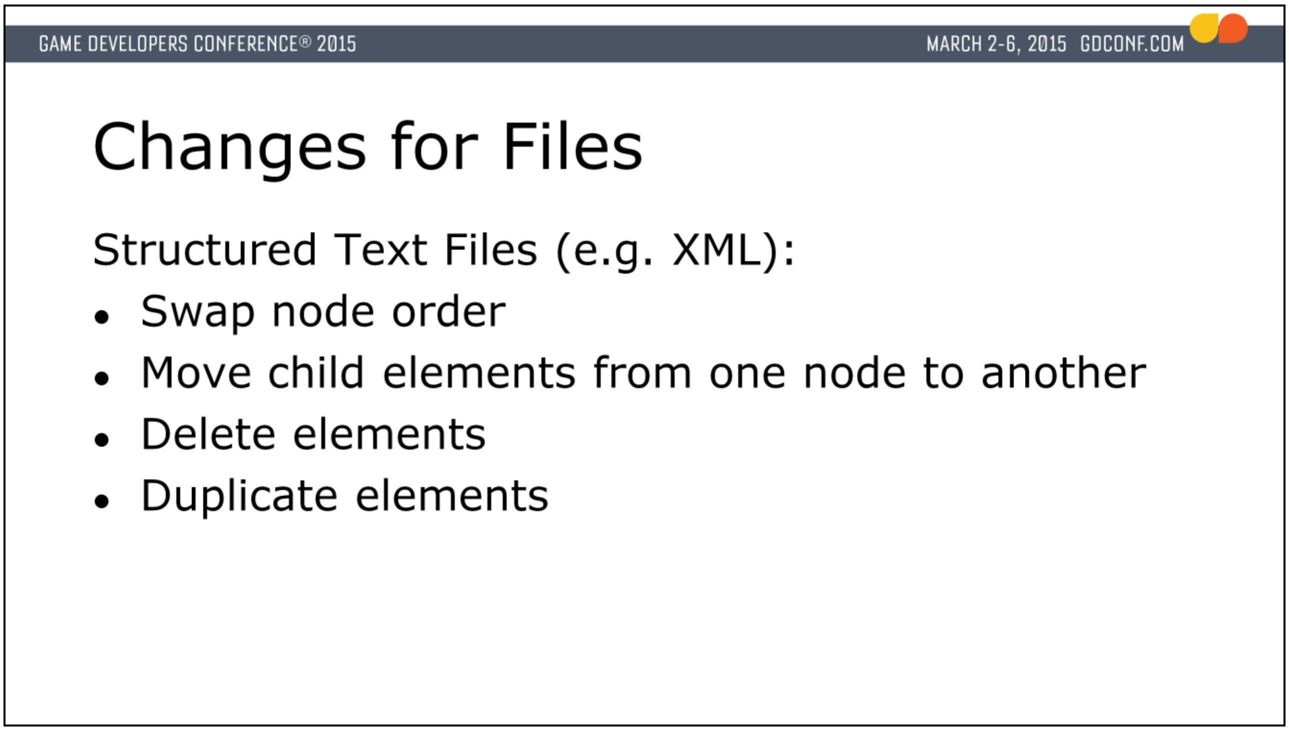

GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Changes for Files

Binary Files:

- Flip bits at random
- Swap bytes at random
  - 0x0, 0xff, 0xfe, 0x7f, 0x80, Current+1, Current-1 all are nice
- Delete a random section of data
- Inject a section of random data

You'll note that these are basically all the changes we were talking about in network fuzzing the payload. Binary data is pretty straightforward to muck with. Don't worry about understanding the content of the file passed anything needed to bypass the weak detections. It is important not to make too many changes to a file (you'll probably only want to make one or two changes, certainly no more than five per iteration).

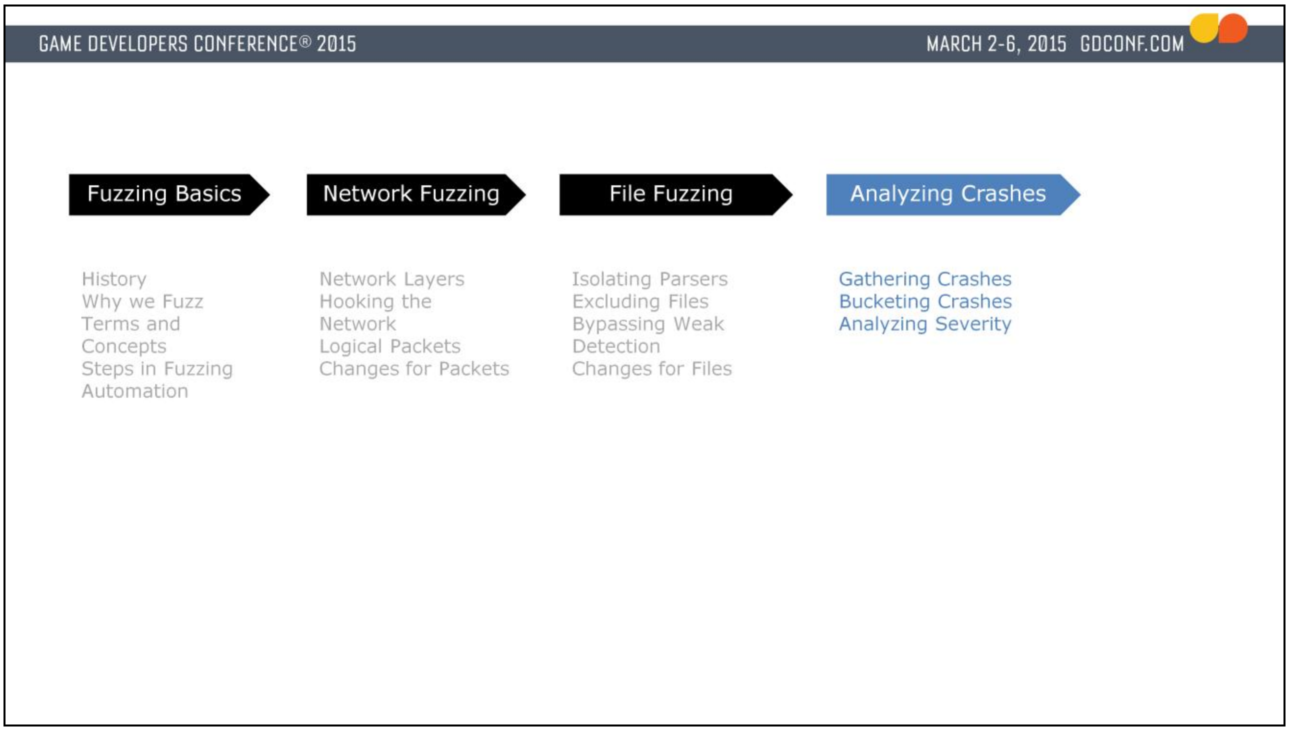


Changes for Files

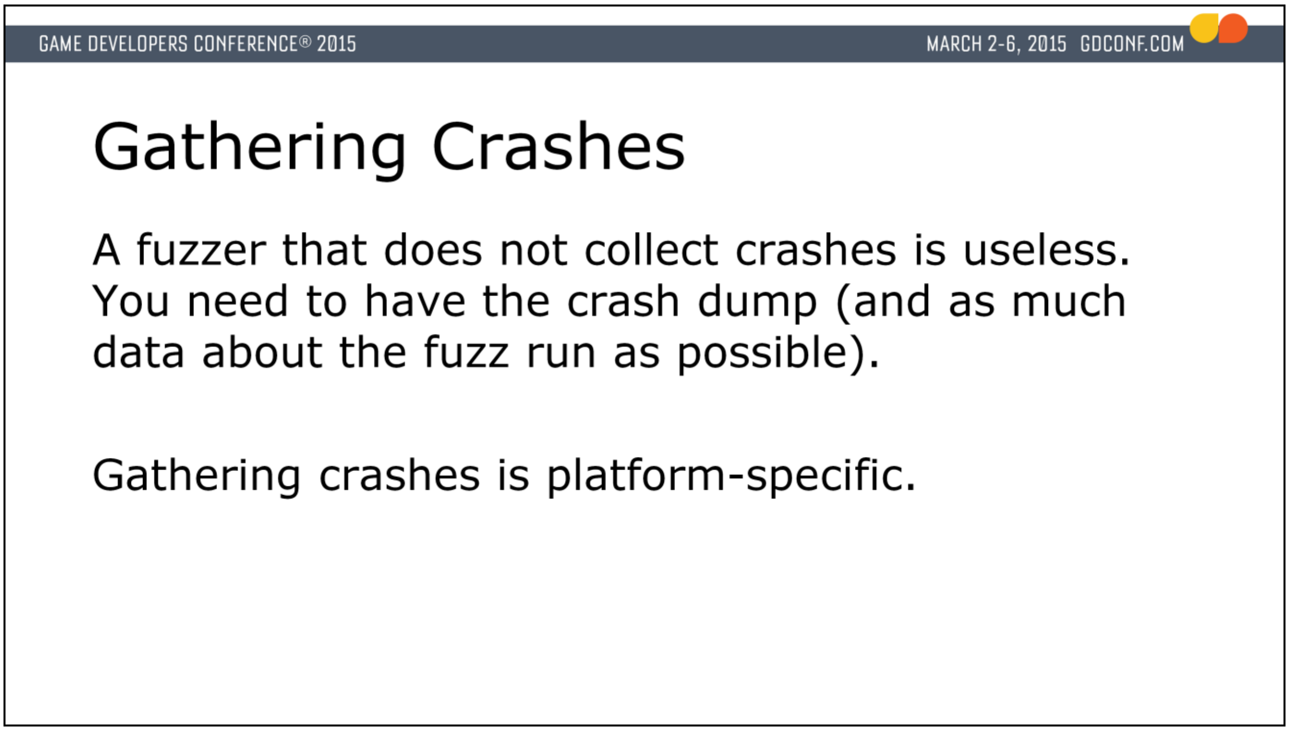
Structured Text Files (e.g. XML):

- Swap node order
- Move child elements from one node to another
- Delete elements
- Duplicate elements

The key to changing something like XML is that you probably don't want to spend a lot of time fuzzing the XML parser itself, you want to fuzz the thing that consumes XML. If you wrote your own XML parser, then throw in some binary changes as well. Again, you want to make a very small number of changes per iteration.



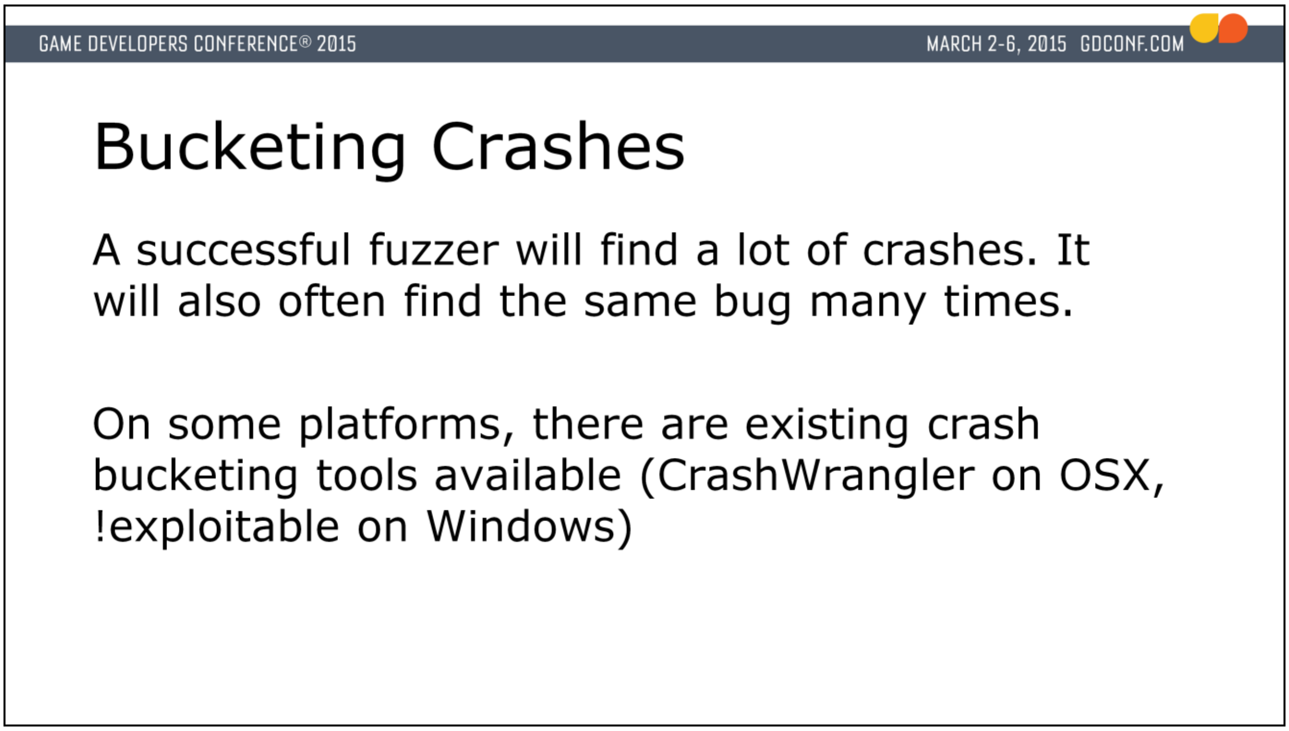
Fuzzing Basics	Network Fuzzing	File Fuzzing	Analyzing Crashes
History Why we Fuzz Terms and Concepts Steps in Fuzzing Automation	Network Layers Hooking the Network Logical Packets Changes for Packets	Isolating Parsers Excluding Files Bypassing Weak Detection Changes for Files	Gathering Crashes Bucketing Crashes Analyzing Severity



Gathering Crashes

A fuzzer that does not collect crashes is useless. You need to have the crash dump (and as much data about the fuzz run as possible).

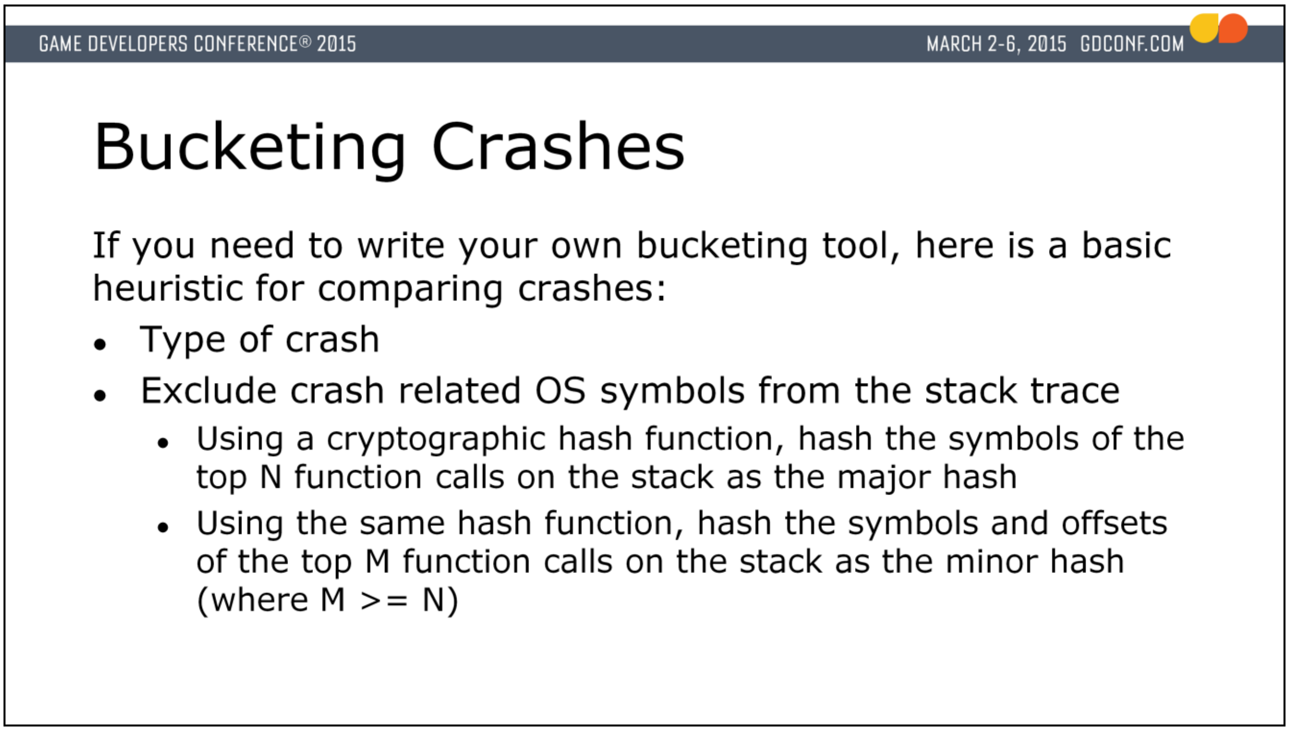
Gathering crashes is platform-specific.



Bucketing Crashes

A successful fuzzer will find a lot of crashes. It will also often find the same bug many times.

On some platforms, there are existing crash bucketing tools available (CrashWrangler on OSX, !exploitable on Windows)

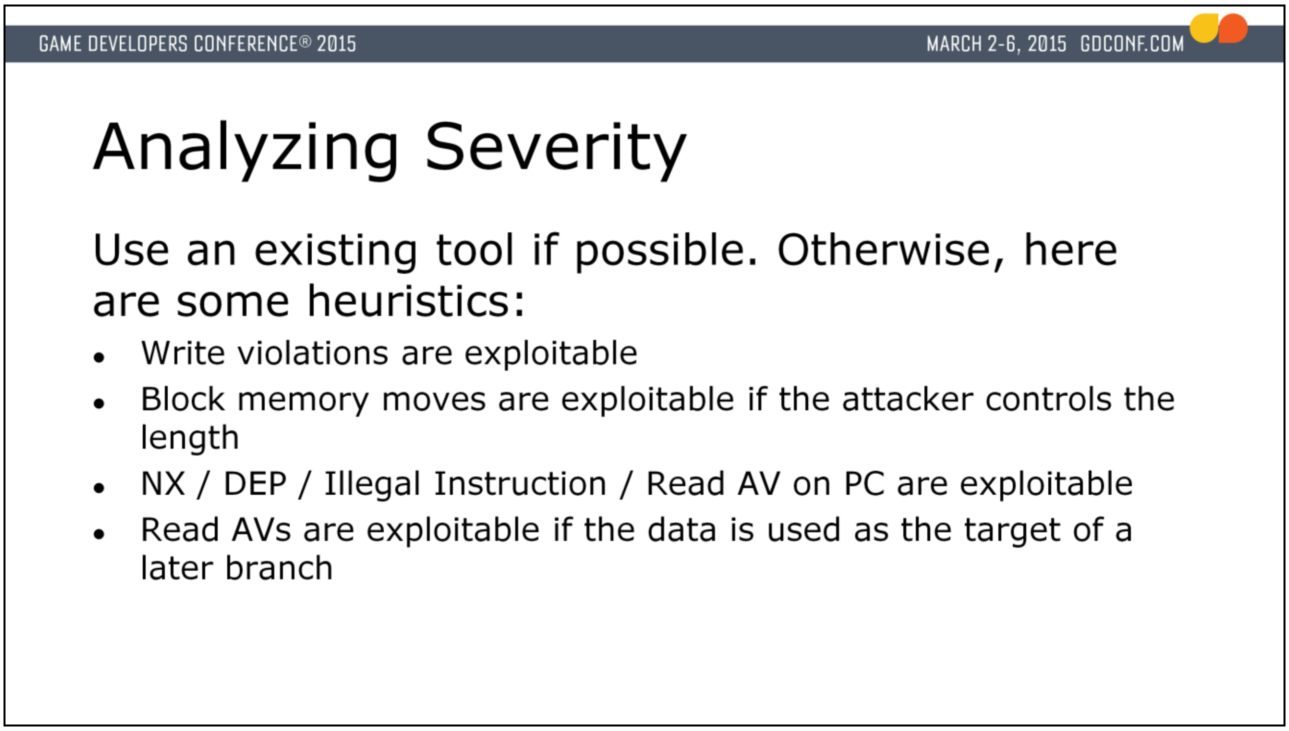


Bucketing Crashes

If you need to write your own bucketing tool, here is a basic heuristic for comparing crashes:

- Type of crash
- Exclude crash related OS symbols from the stack trace
  - Using a cryptographic hash function, hash the symbols of the top N function calls on the stack as the major hash
  - Using the same hash function, hash the symbols and offsets of the top M function calls on the stack as the minor hash (where M >= N)

Crashes with the same major hash are in the same code path. Crashes with the same minor hash are very likely the same bug. Crashes with the same minor hash and type are almost certainly the same bug. Major hashes tend to persist from build to build, Minor hashes are build specific



Analyzing Severity

Use an existing tool if possible. Otherwise, here are some heuristics:

- Write violations are exploitable
- Block memory moves are exploitable if the attacker controls the length
- NX / DEP / Illegal Instruction / Read AV on PC are exploitable
- Read AVs are exploitable if the data is used as the target of a later branch

There are existing tools (again, CrashWrangler or !exploitable) that you could use. !exploitable is an open-source product, so the source is available if you need to write your own tool.

Questions?