Secure Software

Unterschied zwischen Design und Code

Vorgetragen von René Pfeiffer auf den Linuxwochen Eisenstadt 2018.

Sicherheit und Software

- Software
  - Vorläufer: Algorithmen
  - 19. Jahrhundert (Ada Lovelace)
  - Digitales Zeitalter seit 1943 (Z3, Konrad Zuse, Deutsche Luftwaffe)
  - Creeper Test Virus (DEC PDP-10, ARPANET, 1971)
  - Reaper Test Cleaner (jagt und löscht Creeper, 1971)
  - Elk Cloner (Macintosh, 1982) – erster Computer Virus
- Sicherheit
  - Fokus auf Stabilität (keine „crashes“, keine „bugs“, definiertes Verhalten)
  - Historisch unscharf zu definieren (safety/security)
  - Software sollte sich benehmen und sich selbst schützen

Was ist mit den Daten?

- Code sind keine Daten, Daten sind kein Code – selten,
  - weil wir die von-Neumann-Architektur haben
  - und sehr viele Daten interpretiert werden.
- Annahmen
  - Die Welt der Software ist voller Annahmen.
  - Code muss definierten Start haben und Zustand verfolgen.
- Buzzwords und Trends
  - vernebeln die Sicht.
  - Unterschied zwischen Daten-/Code-/Software-/Plattform-/Cloud-Sicherheit?
  - Wie definiert und mißt man „sicher“?
  - Was ist die Metrik für „Fehler“?

Secure Coding

Secure Coding

- Secure Coding
  - Sicherheit implementieren und kritische Fehler vermeiden
  - zu 90% aus Checklisten und Techniken basierend auf „secure mindset“
- Zusätzlicher Code immer notwendig
  - Abfragen, Ausnahmen, Fehlerbehandlung, …
  - Schwächstes Glied bestimmt Sicherheit / Erfolg.
- Verfügbar für Applikationen und Betriebssysteme
  - Nur möglich für aktiven und gewarteten Code.
- Legacy Software fällt oft heraus.
  - Softwarebibliotheken
  - Firmware Blobs
  - Code auf IoT Geräten

Secure Coding ≠ Abschwächung

- Abschwächung nach Versagen/Einbruch
  - Address Space Layout Randomization (ASLR)
  - Buffer Overflow Protection
  - Heap Overflow Protection
  - Stack Overflow Protection
  - No-eXecute (NX) Technologien
    - XD bit (Intel®)
    - Enhanced Virus Protection (AMD™)
- Sandbox
- Container

Secure Coding ≠ Abschwächung (2)

- Airbags für Software
  - Wird gebraucht, wenn Software bereits versagt hat.
  - Kann keine Fehler beheben.
  - Begrenzt Schaden so gut wie möglich.
  - Verhindert Schaden nicht.

Typische Implementation

1. Nehme Code Basis des Projekts
2. Verbinden mit Secure Coding Checklisten
3. Volltextsuche nach „schlechten“ Konstrukten
4. Ersetzen gefundener Codestücke gemäß Checklisten
5. Ergänzen von Verschlüsselung
6. Ergänzen von Athentisierung und Autorisierung (optional)
7. Recompile & Deploy
8. Schwere Fälle: „Nur für interne Netzwerke“ Hinweis anbringen

Secure Design

Secure Design – Vorbereitungen

- Klassifizierung
  - Welche Daten werden verarbeitet? Wie sehen sie *genau* aus?
  - Wer produziert/konsumiert die Daten?
  - Trennung erfolgt aus Basis der Klassifizierung
- Identifizieren der Übergänge
  - Welche grenzen überqueren die Daten?
  - Welche Komponenten kommunizieren mit welchen anderen? Wie? Warum?
  - Wiederholung für Interna: Bibliotheken und Komponenten
- Verfolgen der internen Zustände
  - Kryptographische Algorithmen müssen Zustände komplett beschreiben.
  - Softwareentwicklung kann Anleihe bei Methoden nehmen.

Plattform gut kennen

- Hardware
  - Letztlich landet alles auf der CPU, nur wie?
  - CPU haben Fehler und Eigenheiten
  - x86/x86_64 CPUs nicht kompatibel – Instruction Sets
  - Features verändern Code-Abarbeitung
- Compiler / Interpreter
  - Features sorgen wieder für Veränderung
  - Programmiersprachen haben Eigenheiten
- Betriebssysteme
  - Caches schlecht für sensitive Daten
  - Wie wird mit Speicher verfahren?
- Bibliotheken

Thread Modelling

Secure Design Prinzipien

- Minimieren der Angriffsfläche
- Einführen, unterhalten und verwenden von *Secure Defaults*
- Prinzip minimaler Privilegien
- Raumverteidigung (*Defence in Depth*)
- *Fail Securely*
- Dritten nicht vertrauen (APIs, Komponenten, …)
- Aufgaben verteilen/verkapseln
- Kein *Security by Obscurity*
- *Keep It Simple*

Formale Methoden

- Formale Methoden (FM) verwenden Mathematik, um Code ganz zu erfassen
  - Spezifikation, Design, Überprüfung wird mathematisiert
  - Simulation wird berechnet/aufgelöst
- Mathematik meint
  - „*specifications are well-formed statements in a mathematical logic and that the formal verifications [if any] are rigorous deductions in that logic*“
  - Abdeckung aller Zustände & Fälle
- FM benötigt spezielle Tools und Beschreibung
  - FM-basierte Beschreibungssprache
  - Werkzeuge zur Überprüfung
  - Beweise durch höhere Mathematik

Kryptografie

- Kryptografie ist Teil moderner Software
  - Protokolle, Datentransport, Datenspeicherung
  - Zufall – Schlüssel, Sessions, Tokens, Unique Identifiers, …
  - Public Key Infrastructure (PKI)
- Hash Funktionen
  - verfügbar in *crypto* und *non-crypto*
  - oft verwendet
- Authentifizierung
  - Verschlüsselung wird nie/selten signiert
  - Verschlüsselung und Schlüssel sind Identitäten egal

Secure Design

- Secure Design muß Teil des *Konzepts* sein
  - Secure Design beginnt vor der ersten Zeile Code
  - Security Entscheidungen müssen eingebaut sein
- Secure Design verwendet Secure Coding automatisch
  - Datenvalidierung immer präsent – Mißtrauen internen Komponenten
  - Unsichere Konstrukte dürfen nicht Teil des Codes sein
  - Gilt auch für Datenformate und Protokolle!
- Secure Design ist Teil aller Phasen
  - Konzeptionierung, Design (auch UI!), Coding, Testen, Deployment, …
  - Daten – alle Daten, die verarbeitet und produziert werden

Beschränkungen

- Defekte Architektur bleibt defekt
  - CPU Bugs (Meltdown, Spectre), Klartextprotokoll, FTP, …
  - Gilt auch für Lieblings-Toolchain/-sprache/-IDE
  - Verschweigen schützt niemanden
- Workarounds beginnen 10 Minuten nach dem Design
  - Zu komplexe Datenformate (PDF, Office Dokumente, aktive Inhalte, …)
  - Sichere Protokolle haben Voraussetzungen (PKI, Crypto, …)
  - Zusätzliche Komponenten notwendig
- Abhilfe: Gleich mit Fehlern beginnen!
  - Misstrauen!
  - Re-validieren!

Fehler einbauen

- Instabilitäten lassen sich durch
  - manipulierte Daten oder
  - logische Konstrukte (Debug Code)

  einbauen.
- Code muss Fehler richtig behandeln
  - kein undefiniertes Verhalten
  - Validierungscode ist gefordert

Linux® Kernel Fault Injection

- Linux® Kernel verwendet künstliche Fehler
- Funktionsaufrufe sabotieren
  - Konfigurierbare Intervalle und Wahrscheinlichkeit
  - Funktioniert mit Storage, Dateisystemen und Syscalls
- Verwendet für Debugging / Development
- Leicht zu verwenden
  - Teil des Standard Linux® Kerns
  - Konfigurierbar via debugfs/procfs

Beispiel: Fault Injection

- `mount debugfs /debug -t debugfs`
- `cd /debug/fail_make_request`
- `echo 10 > interval # interval`
- `echo 100 > probability # 100% probability`
- `echo -1 > times # how many times: -1 means no limit`
- `mount -t logfs /dev/sdb1 /mnt/2`
- `echo 1 > /sys/block/sdb/sdb1/make-it-fail`

Beispiel: Fault Injection

```
FAULT_INJECTION: forcing a failure
[] show_trace_log_lvl+0x1a/0x2f
[] show_trace+0x12/0x14
[] dump_stack+0x15/0x17
[] should_fail+0x113/0x144
[] generic_make_request+0x167/0x2c0
[] submit_bio+0xd7/0xdf
[] submit_bh+0xbc/0xd9
[] block_read_full_page+0x26c/0x27c
[] blkdev_readpage+0xf/0x11
[] read_cache_page_async+0x96/0x129
[] read_cache_page+0x12/0x42
[] bdev_read+0x5d/0xcd
[] scan_segment+0x16e/0x331
[] logfs_scan_pass+0x2d/0x71
[] logfs_gc_pass+0x44/0x3f3
[] logfs_get_wblocks+0x21/0x2f
[] logfs_write_buf+0x22/0x30b
[] logfs_commit_write+0xed/0x101
[] generic_file_buffered_write+0x3a9/0x522
[] __generic_file_aio_write_nolock+0x43b/0x471
[] generic_file_aio_write+0x64/0xc1
[] do_sync_write+0xc4/0x101
[] vfs_write+0xaf/0x138
[] sys_write+0x3d/0x61
[] sysenter_past_esp+0x5f/0x99
=====================
Buffer I/O error on device sdb1, logical block 35497
------------[ cut here ]------------
kernel BUG at fs/logfs/gc.c:88!
invalid opcode: 0000 [#1]
PREEMPT
Modules linked in: iptable_nat nf_nat ipt_ULOG ipt_recent nf_conntrack_ipv4 xt_state nf_conntrack nfnetlink ipt_REJECT xt_tcpudp iptable_filter ip_tables x_tables capability usblp commoncap loop dm_mod video thermal sbs fan container dock battery ac floppy cpufreq_conservative cpufreq_powe
CPU: 0
EIP: 0060:[] Tainted: P VLI
EFLAGS: 00010282 (2.6.22.5-1 #2)
EIP is at scan_segment+0x172/0x331
eax: fffffffb ebx: c02cfa98 ecx: c1809d80 edx: 00000000
esi: 08aa0000 edi: c749fbdc ebp: c749fc1c esp: c749fb8c
ds: 007b es: 007b fs: 0000 gs: 0033 ss: 0068
Process rsync (pid: 3940, ti=c749e000 task=d79574e0 task.ti=c749e000)
Stack: 0000001c c749fbdc 0000031a 00000000 0000899c f695dff4 00000013 00000000
00000455 dd4c1c00 00020000 00000000 022cfa98 00009114 00009114 00000000
00000455 00000000 0000101c d7e7c800 5ccc4658 00040010 00000000 13000000
Call Trace:
[] show_trace_log_lvl+0x1a/0x2f
[] show_stack_log_lvl+0x9b/0xa3
[] show_registers+0x1d7/0x30c
[] die+0xfe/0x1d6
[] do_trap+0x89/0xa2
[] do_invalid_op+0x88/0x92
[] error_code+0x6a/0x70
[] logfs_scan_pass+0x2d/0x71
[] logfs_gc_pass+0x44/0x3f3
[] logfs_get_wblocks+0x21/0x2f
[] logfs_write_buf+0x22/0x30b
[] logfs_commit_write+0xed/0x101
[] generic_file_buffered_write+0x3a9/0x522
[] __generic_file_aio_write_nolock+0x43b/0x471
[] generic_file_aio_write+0x64/0xc1
[] do_sync_write+0xc4/0x101
[] vfs_write+0xaf/0x138
[] sys_write+0x3d/0x61
[] sysenter_past_esp+0x5f/0x99
=====================
Code: 00 00 00 0f a5 f7 d3 e6 f6 c1 20 0f 45 fe 0f 45 f0 8b 45 94 89 f9 89 f2 8d 7d c0 03 55 a8 13 4d ac 89 7c 24 04 ff 13 85 c0 74 04 <0f> 0b eb fe 8d 45 c0 b9 1c 00 00 00 ba ff 00 00 00 e8 82 55 00
EIP: [] scan_segment+0x172/0x331 SS:ESP 0068:c749fb8c
```

Fuzzing

- Fuzzing verwendet Zufallszahlen für Tests
  - Technik seit den 1950er bekannt (defekte Lochkarten)
  - Forschungthema seit 1981
- verschiedene Strategien
  - komplett zufällige Daten
  - strukturiert-zufällige Daten
  - strukturiert-zufällige und code-zufällige Daten
- Black/White Box Ansatz
- Standardwerkzeug für Softwareentwicklung

Zufall oder nicht?

- völlig zufälliges Testen
  - kann Code Zweige verpassen, weil Daten nicht erkannt werden,
  - generiert sehr viele Daten.
  - Geeignet für Stresstests
- Struktur und Zufall mischen sehr effektiv
  - Zielt auf die Validationslogik
  - Effektiver als oberflächlicher Test

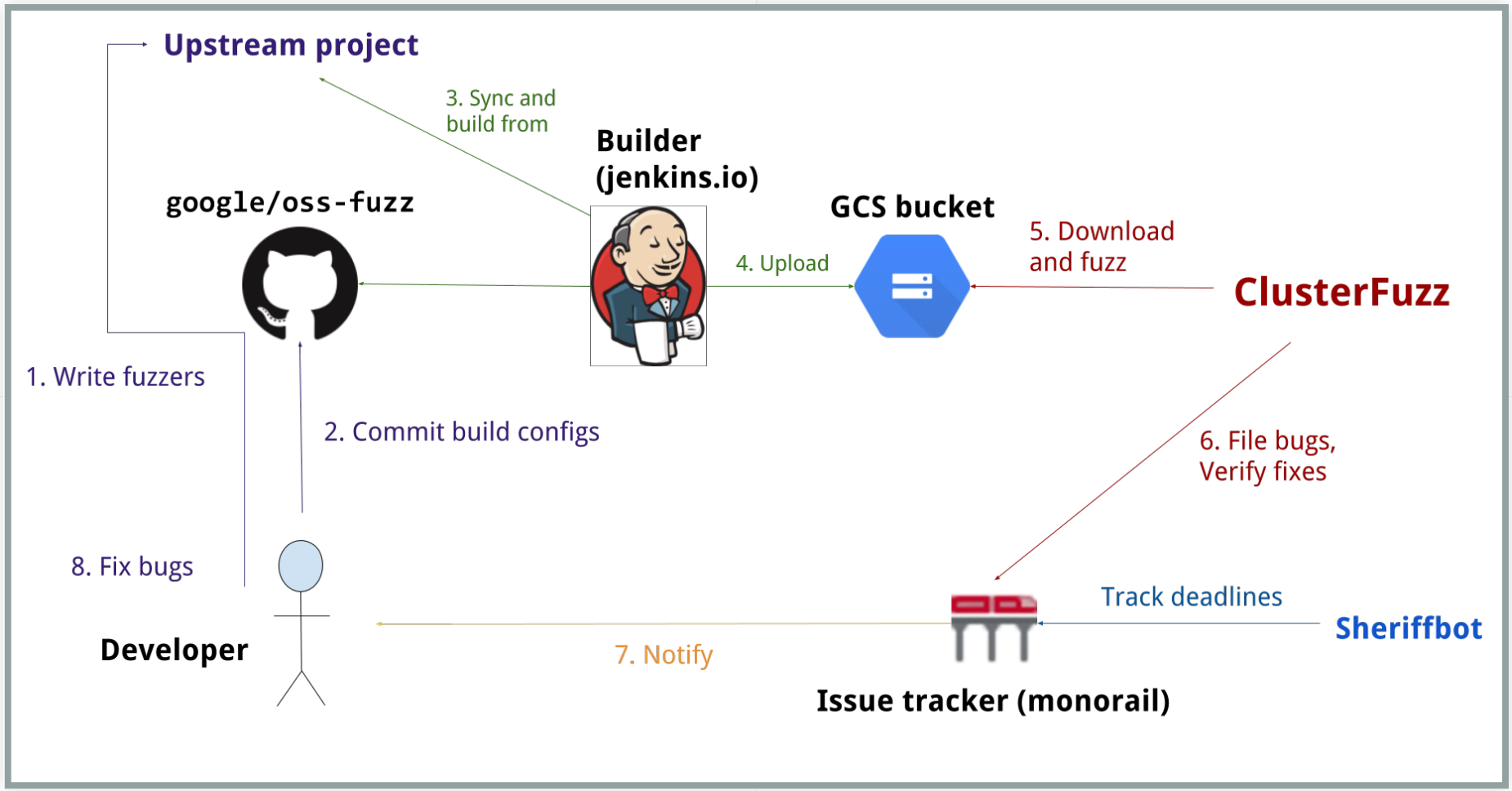
American Fuzzy Lop (AFL)

- AFL ist Fuzzing Tool für C/C++
  - Synthetisiert Dateiformate (analysiert Syntax)
  - Minimiert Testfälle
  - Kann Abstürze gezielt „erforschen“
- Benutzt Beispieldaten als Basis
  - Speichert modifizierte Versionen der Daten als Referenz
  - Verwendet Wörterbücher und Statistik
- Abstürze sind nicht das Ziel
- AFL braucht Ressourcen
  - Schnelle und fehlerfreie CPU
  - Milliarden Schreib-/Lesezyklen im Dateisystem

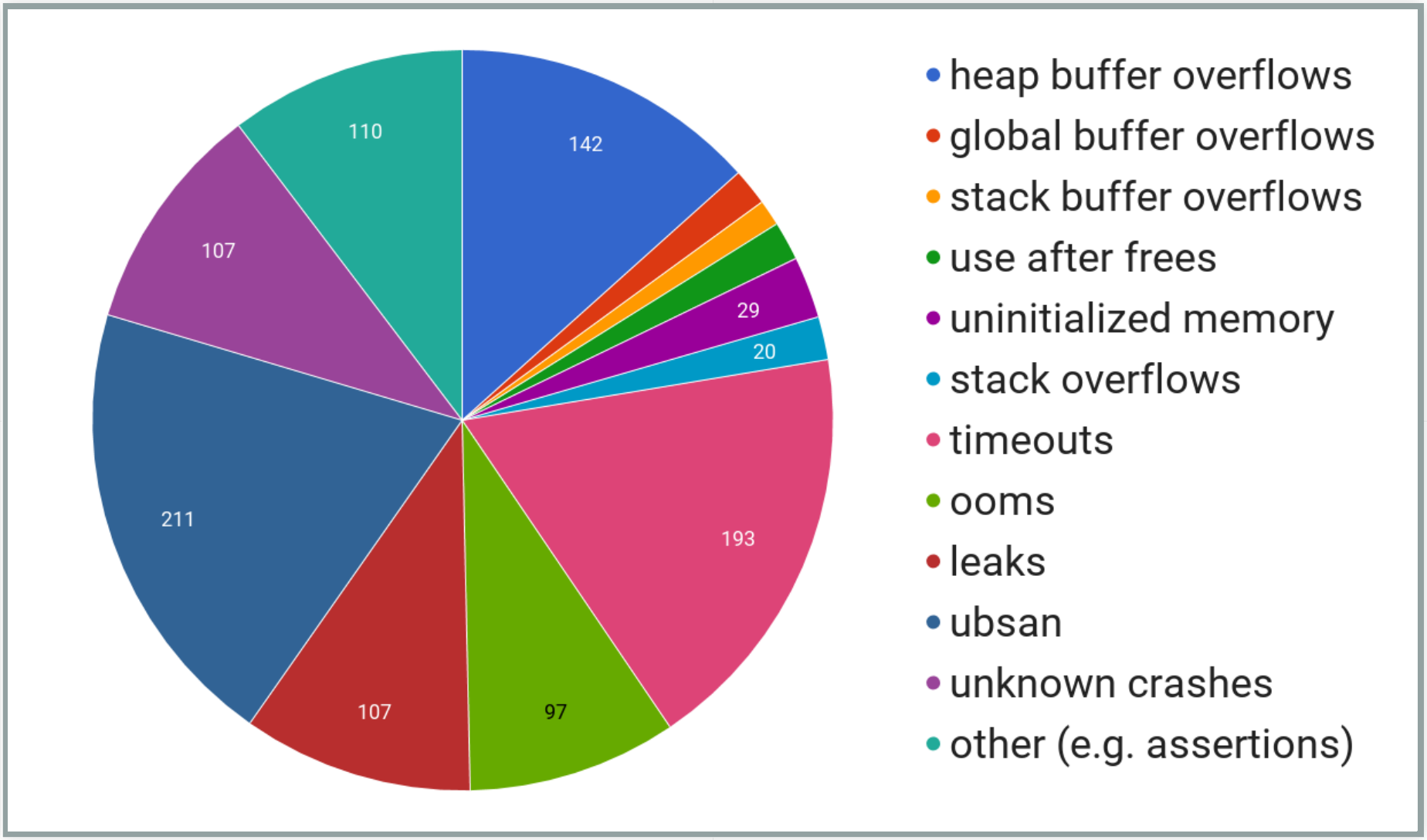
AFL Ergebnisse

- Two bignum libraries give different output for the same fuzzer-generated input.
- Image library produces different outputs when asked to decode the same image repeatedly.
- Serialisation/deserialisation library fails to produce stable outputs for fuzzer-generated data structures.
- Compression library produces inconsistent output for fuzzer-generated input (compress-decompress cycle with comparison).
- AFL found the Heartbleed bug (again) by letting OpenSSL talk to itself.

Google OSS Fuzz Plattform



Ergebnisse: Google OSS Fuzz



Zusammenfassung

- Secure Coding ≠ Secure Design.
- Secure Design ≠ Design.
  - Secure Design hat zusätzliche Anforderungen
  - Erfordert Integration mit Softwareentwicklung
- Nicht alle Applikationen werden Secure Design/Coding verwenden.
- Hardware, Bibliothek, Programmiersprache ist defekt.
- Fehler sind Bestandteil von Softwareprojekten.
- Wieso nicht die Applikation gleich ins Netz stellen?

Noch Fragen?


string sInput;
int iLength, iN;
double dblTemp;
bool again = true;
while (again) {
iN = -1;
again = false;
getline(cin, sInput);
system("cls");
stringstream(sInput) >> dblTemp;
iLength = sInput.length();
if (iLength < 4) {
again = true;
continue;
} else if (sInput[iLength - 3] != '.') {
again = true;
continue;
} while (++iN < iLength) {
if (isdigit(sInput[iN])) {
continue;
} else if (iN == (iLength - 3)) {

Über die SEC4YOU

Die SEC4YOU Advanced IT-Audit Services GmbH ist ein produktunabhängiges Prüfungs- und Beratungsunternehmen im Bereich der IT-/Informationssicherheit. Durch die langjährige Erfahrung in IT-Audits bereiten wir Kunden praxisorientiert und kostenoptimiert auf die wachsenden IT-Compliance Anforderungen vor. Wir unterstützen bei der Einführung und dem Betrieb von ISMS nach ISO-27001 und interner IT-Kontrollsysteme (IKS) mit Beratung und IT-Audits unter Berücksichtigung der gesetzlichen Anforderungen, z.B. der EU-DSGVO. Zahlreiche Referenzen können wir aus den Fachbereichen Interne Revision, Wirtschaftsprüfung, IT, Datenschutz, sowie Risiko- und Compliancemanagement vorweisen.

Über den Autor

René Pfeiffer ist selbstständiger Systemadministrator und Vortragender an der Fachhochschule Technikum Wien im Bereich Computer- und Datensicherheit. Mit über 15 Jahren Berufs- und 30 Jahren Computererfahrung sowie einem akademischen Hintergrund in theoretischer Physik verbindet er in Schulungen und Projekten erfolgreich Theorie und Praxis.

Seine Themenschwerpunkte liegen im Bereich IT Administration, Aufbau sicherer Infrastrukturen (Host-/Netzwerkbereich), sichere Kommunikation in Organisationen und Infrastruktur (VPN Technologien, Nachrichtensysteme), Wireless Security, technische Dokumentation, Betreuung von Forschungsarbeiten in der IT Security, technischem Auditing und Evaluierung von Software.

Herr Pfeiffer hält seit 2000 jährlich Fachvorträge und Schulungen auf Tagungen und Seminaren (Institute for International Research (I.I.R.), Business Circle, Confare, Linuxwochen Österreich, SAE Institute Wien, DeepSec In-Depth Security Conference, Bundesverband Sicherheitspolitik an Hochschulen, Kundenveranstaltungen).

Bildquellen / Artikel

Die Quellen sind zusätzlich oder teilweise statt in dieser Liste an den Bildern selbst angebracht.

Kontaktmöglichkeiten

- E-Mail: *rene.pfeiffer@sec4you.com*
- PGP/GPG: `0xA60F81C8CA93AD2B`
- Mobil: `+43.676.5626390` (Signal verfügbar)
- Threema: `4WFYBWCJ`