Cyber Grand Shellphish

DEFCON 24
August 7, 2016 · Track 2 - 3pm
HEX on the beach
Success.
DARPA Competitions

Self-driving Cars

Robots
The DARPA Cyber Grand Challenge

Programs!
analyze

pwn

patch
analyze

pwn

patch
- Linux-inspired environment, with only 7 syscalls
  ■ transmit / receive / fdwait (~select)
  ■ allocate / deallocate
  ■ random
  ■ terminate

- No need to model the POSIX API!

- Otherwise real(istic) programs.
analyze

pwn

patch
- No filesystem -> no flag?

- CGC Quals: crash == exploit

- CGC Finals: two types of exploits
  1. "flag overwrite": set a register to X, crash at Y
  2. "flag read": leak the "secret flag" from memory
analyze

pwn

patch
int main() { return 0; }

fails functionality checks...

signal(SIGSEGV, exit)

no signal handling!

inline QEMU-based CFI?

performance penalties...
Mechanical Phish (CQE)

A completely autonomous system

• Patch
• Crash
Mechanical Phish (CFE)

Completely autonomous system

- Patch
- Crash
- Exploit
The DARPA Cyber Grand Challenge
The CGC Final Event (CFE)

• The competition is divided in rounds (96), with short breaks between rounds
• The competition begins: The system provides a set of Challenge Binaries (CBs) to the teams’ CRSs
  – Each CB provides a service (e.g., an HTTP server)
  – Initially, all teams are running the same binaries to implement each service
• For each round, a score for each (team, service) tuple is generated
The CGC Final Event (CFE)

\[
\sum_{i=0}^{CB} \text{Availability} \times \text{Security} \times \text{Evaluation}
\]

- Availability: how badly did you fuck up the binary?
- Security: did you defend against \textit{all} exploits?
- Evaluation: how many n00bs did you pwn?

- When you are shooting blindfolded automatic weapons, it’s easy to shoot yourself in the foot...
Code Freeze?

cao  4:01 PM
farnsworth has been freezed
  all outstanding merge requests have been merged in

mike_pizza  4:01 PM
holy shit

cao  4:02 PM
set the channel topic: meister and farnsworth are in code freeze
oops!
<table>
<thead>
<tr>
<th>Commit Message</th>
<th>Author</th>
<th>Author Date</th>
<th>Commit ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>God please forgive me for this commit</td>
<td>Francesco Disperati</td>
<td>22 days ago</td>
<td>72a44980</td>
</tr>
<tr>
<td>Fixes</td>
<td>Francesco Disperati</td>
<td>22 days ago</td>
<td>18849985</td>
</tr>
<tr>
<td>Disable IDSSSubmitter</td>
<td>Francesco Disperati</td>
<td>23 days ago</td>
<td>460fc02c</td>
</tr>
<tr>
<td>Capitalize constant</td>
<td>Francesco Disperati</td>
<td>23 days ago</td>
<td>60cb8fe0</td>
</tr>
<tr>
<td>pass patchtype to PatcherexJob</td>
<td>Antonio Bianchi</td>
<td>23 days ago</td>
<td>160a89d4</td>
</tr>
</tbody>
</table>

15 Jul, 2016 20 commits
Tue 2 Aug, 23:54
~15 hours before access shutdown
Farnsworth

Object-relational model for database:
- What CS are fielded this round?
- Do we have crashes?
- Do we have a good patch?
- ...

Our ground truth and the only component reasonably well tested*

* 69% coverage
Meister

Job scheduler:

• Looks at game state
• Asks creators for jobs
• Schedules them based on priority
On the Shoulders of Giants

Z3

python

ubuntu

angr

AFL

kubernetes

Unicorn Engine

pypy

PostgreSQL

VEX

Capstone Engine

docker
• Framework for the analysis of binaries, developed at UCSB
• Supports a number of architectures
  – x86, MIPS, ARM, PPC, etc. (all 32 and 64 bit)
• Open-source, free for commercial use (!)
  – http://angr.io
  – https://github.com/angr
  – angr@lists.cs.ucsb.edu
Concolic Execution

Automatic Exploitation

Patching
Fuzzing

• Fuzzing is an automated procedure to send inputs and record safety condition violations as crashes
  – Assumption: crashes are potentially exploitable

• Several dimensions in the fuzzing space
  – How to supply inputs to the program under test?
  – How to generate inputs?
  – How to generate more “relevant” crashes?
  – How to change inputs between runs?

• Goal: maximized effectiveness of the process
Gray/White-box Fuzzing

Bugs (0-day)

Application Under Analysis

Input Generator

Feedback

Fuzzing Infrastructure

Crash

Crash Database
How do we find crashes?

Network Traffic

Symbolic Execution

Fuzzing

"Uncrasher"
Fuzzing: American Fuzzy Lop
x = int(input())
if x >= 10:
    if x < 100:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"

Let's fuzz it!

1 ⇒ "You lose!"
593 ⇒ "You lose!"
183 ⇒ "You lose!"
4 ⇒ "You lose!"
498 ⇒ "You lose!"
42 ⇒ "You win!"
x = int(input())
if x >= 10:
    if x^2 == 152399025:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"

Let's fuzz it!

1 ⇒ "You lose!"
593 ⇒ "You lose!"
183 ⇒ "You lose!"
4 ⇒ "You lose!"
498 ⇒ "You lose!"
42 ⇒ "You lose!"
3 ⇒ "You lose!"
........
57 ⇒ "You lose!"
- Very fast!

- Very effective!

- Unable to deal with certain situations:
  - magic numbers
  - hashes
  - specific identifiers
x = input()
if x >= 10:
    if x % 1337 == 0:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```python
x = input()
if x >= 10:
    if x % 1337 == 0:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```
Driller = AFL + angr

Fuzzing
- good at finding solutions for general inputs

Symbolic Execution
- good at finding solutions for specific inputs
Driller

Test Cases
Driller

“Cheap” fuzzing coverage

Test Cases

“χ”

“γ”
Driller

“Cheap” fuzzing coverage

Dynamic Symbolic Execution

Test Cases

“Χ”

“Υ”
Driller

“Cheap” fuzzing coverage

Dynamic Symbolic Execution

New test cases generated

Test Cases

“X”

“Y”

“CGC_MAGIC”
“Cheap” fuzzing coverage

Dynamic Symbolic Execution

New test cases generated

Test Cases

“X”

“Y”

“CGC_MAGIC”

“CGC_MAGICY”
typedef struct component {
    char name[32];
    int (*do_something)(int arg);
} comp_t;

comp_t *initialize_component(char *cmp_name) {
    int i = 0;
    struct component *cmp;

    cmp = malloc(sizeof(struct component));
    cmp->do_something = sample_func;

    while (*cmp_name)
        cmp->name[i++] = *cmp_name++;

    cmp->name[i] = ‘\0’;
    return cmp;
}

x = get_input();
cmp = initialize_component(x);
cmp->do_something(1);
Auto Exploitation - Simplified

Turning the state into an **exploited** state

```
angr
assert state.se.symbolic(state.regs.pc)
```

Constrain **buffer** to contain our shellcode

```
angr
buf_addr = find_symbolic_buffer(state, len(shellcode))
mem = state.memory.load(buf_addr, len(shellcode))
state.add_constraints(mem == state.se.bvv(shellcode))
```
Auto Exploitation - Simplified

Constrain **PC** to point to the buffer

```python
    angr

    state.se.add_constraints(state.regs.pc == buf_addr)
```

**Synthesize!**

```python
    angr

    exploit = state.posix.dumps(0)
```
Auto Exploitation - Simplified

- Vulnerable Symbolic State (PC hijack)
- Constraints to add shellcode to the address space
- Constraints to make PC point to shellcode

Exploit
Detecting Leaks of the Flag Page

• Make only the flag page symbolic

• Everything else is completely concrete
  – Can execute most basic block with the Unicorn Engine!

• When we have idle cores on the CRS, trace all our testcases

• Solved DEFCON CTF LEGIT_00009 challenge
Patcherex

Patching Techniques:
- Stack randomization
- Return pointer encryption
- ...

Patches:
- Insert code
- Insert data
- ...

Patching Backend:
- Detour
- Reassembler
- Reassembler Optimized
Adversarial Patches 1/2

Detect QEMU

xor eax, eax
inc eax
push eax
push eax
push eax
fld TBYTE PTR [esp]
fsqrt
Adversarial Patches 2/2

Transmit the flag
- To stderr!

Backdoor

- hash-based challenge-response backdoor
- not “cryptographically secure” → good enough to defeat automatic systems
Generic Patches

Return pointer encryption

Protect indirect calls/jmps

Extended Alloc allocations

Randomly shift the stack (ASLR)

Clean uninitialized stack space
Targeted Patches

Qualification event → avoid crashes!
Targeted Patches

Final event →
Reassembler & Optimizer

- Prototypes in 3 days
  
angr is awesome!!

- A big bag of tricks integrated, which worked out
CGC CFE Statistics 1/3

- 82 Challenge Sets fielded
- 2442 Exploits generated
- 1709 Exploits for 14/82 CS with 100% Reliability
- Longest exploit: 3791 lines of C code
- Shortest exploit: 226 lines of C code
- crackaddr: 517 lines of C code
CGC CFE Statistics 2/3

100% reliable exploits generated for:

- YAN01_000{15,16}
- CROMU_000{46,51,55,65,94,98}
- NRFIN_000{52,59,63}
- KPRCA_00{065,094,112}

Rematch Challenges:

- SQLSlammer (CROMU_00094)
- crackaddr (CROMU_00098)
Vulnerabilities in CS we exploited:

- CWE-20 Improper Input Validation
- CWE-119 Improper Restriction of Operations within the Bounds of a Memory Buffer
- CWE-121: Stack-based Buffer Overflow
- CWE-122: Heap-based Buffer Overflow
- CWE-126: Buffer Over-read
- CWE-131: Incorrect Calculation of Buffer Size
- CWE-190: Integer Overflow or Wraparound
- CWE-193 Off-by-one Error
- CWE-201: Information Exposure Through Sent Data
- CWE-202: Exposure of Sensitive Data Through Data Queries
- CWE-291: Information Exposure Through Sent Data
- CWE-681: Incorrect Conversion between Numeric Types
- CWE-787: Out-of-bounds Write
- CWE-788: Access of Memory Location After End of Buffer
Human augmentation...

Awesome:
- CRS assisted with 5 exploits
- Human exploration -> CRS exploitation
- Backdoors!

Tough:
- API incompatibilities are brutal
- Computer programs are brittle
Open source all the code!
Stay in touch!

twitter: @Shellphish
email: team@shellphish.net or cgc@shellphish.net
irc: #shellphish on freenode

CRS chat: #shellphish-crs on freenode
angr chat: #angr on freenode
Backup
Conclusions

- Automated vulnerability analysis and mitigation is a growing field
- The DARPA CGC Competition is pushing the limits of what can be done in a self-managed, autonomous setting
- This is a first of this kind, but not the last
- ... to the singularity!
Self-Managing Hacking

• Infrastructure availability
  – (Almost) No event can cause a catastrophic downtime
    • Novel approaches to orchestration for resilience

• Analysis scalability
  – Being able to direct efficiently (and autonomously) fuzzing and state exploration is key
    • Novel techniques for state exploration triaging

• Performance/security trade-off
  – Many patched binaries, many approaches: which patched binary to field?
    • Smart approaches to security performance evaluation
Hacking Binary Code

• Low abstraction level
• No structured types
• No modules or clearly defined functions
• Compiler optimization and other artifacts can make the code more complex to analyze
• WYSIWYE: What you see is what you execute
Finding Vulnerabilities

Human

Semi-Automated

Fully Automated
Manual Vulnerability Analysis

• “Look at the code and see what you can find”
• Requires substantial expertise
  – The analysis is as good as the person performing it
• Allows for the identification of complex vulnerabilities (e.g., logic-based)
• Expensive, does not scale
Tool-Assisted Vulnerability Analysis

• “Run these tools and verify/expand the results”

• Tools help in identifying areas of interest
  – By ruling out known code
  – By identifying potential vulnerabilities

• Since a human is involved, expertise and scale are still issues
Automated Vulnerability Analysis

- “Run this tool and it will find the vulnerability”
  - ... and possibly generate an exploit...
  - ...and possibly generate a patch
- Requires well-defined models for the vulnerabilities
- Can only detect the vulnerabilities that are modeled
- Can scale (not always!)
- The problem with halting...
Vulnerability Analysis Systems

- Usually a composition of static and dynamic techniques
- Model how attacker-controlled information enter the system
- Model how information is processed
- Model a number of unsafe conditions
Static Analysis

• The goal of static analysis techniques is to characterize all possible run-time behaviors over all possible inputs without actually running the program
• Find possible bugs, or prove the absence of certain kinds of vulnerabilities
• Static analysis has been around for a long while
  – Type checkers, compilers
  – Formal verification
• Challenges: soundness, precision, and scalability
Example Analyses

• Control-flow analysis: Finds and reasons about all possible control-flow transfers (sources and destinations)
• Data-flow analysis: Reasons about how data flows within the program
• Data dependency analysis: Reasons about how data influences other data
• Points-to analysis: Reasons about what values can pointers take
• Alias analysis: Determines if two pointers might point to the same address
• Value-set analysis: Reasons about what are the set of values that variables can hold
Dynamic Analysis

- Dynamic approaches are very precise for particular environments and inputs
  - Existential proofs
- However, they provide no guarantee of coverage
  - Limited power
Example Analyses

• Dynamic taint analysis: Keeps track of how data flows from sources (files, network connections) to sinks (buffers, output operations, database queries)

• Fuzzing: Provides (semi)random inputs to the program, looking for crashes

• Forward symbolic execution: Models values in an abstract way and keeps track of constraints
The Shellphish CRS: Mechanical Phish

vulnerable binary

Automatic Vulnerability Finding

Automatic Exploitation

exploit

Automatic Testing

patched binary

Automatic Vulnerability Patching

proposed patches

proposed exploits
Interactive, Online CTFs

- Very difficult to organize
- Require substantial infrastructure
- Difficult to scale
- Focused on both attacking and defending in real time
- From ctftime.org: 100+ events listed
- Online attack-defense competitions:
  - UCSB iCTF 13 editions
  - RuCTF 5 editions
  - FAUST 1 edition
CTFs Are Playgrounds...

- For people (hackers)
- For tools (attack, defense)
- But can they be used to advance science?
DECREE API

- `void _terminate(unsigned int status);`
- `int allocate(size_t length, int prot, void **addr);`
- `int deallocate(void *addr, size_t length);`
- `int fdwait(int nfds, fd_set *readfds, fd_set *writefds, struct timeval *timeout, int *readyfds);`
- `int random(void *buf, size_t count, size_t *rnd_bytes);`
- `int receive(int fd, void *buf, size_t count, size_t *rx_bytes);`
- `int transmit(int fd, const void *buf, size_t count, size_t *tx_bytes);`
Soundness and Completeness

Actual run-time behaviors

$P$
Soundness and Completeness

- Over-approximation (sound)
- Actual run-time behaviors

\( P \)
Soundness and Completeness

More precise over-approximation (sound)

Actual run-time behaviors

$P$
Soundness and Completeness

- Actual run-time behaviors
- Under-approximation (complete)
  - $P$
Soundness and Completeness

Actual run-time behaviors

Unsound, incomplete analysis
Hidden
Changed with "All the things" meme

Open the source!
Human + Machine = WIN!

OMG, can’t do stairs?!
Simulation For Team Shellphish

• **R00:** Competition fields CB1, CB2, CB3
• **R01:** CRS generates PoV1, RB2
  – Points for round 00:
    • (CB1, CB2, CB3): Availability=1, Security=2, Evaluation=1 → Score = 2
    • Total score: 6
• **R02:** Competition fields CB1, RB2, CB3
  – Points for round 01
    • CB1: Availability=1, Security=1, Evaluation= 1+(6/6) → Score = 2
    • RB2: 0
    • CB3: Availability=1, Security=2, Evaluation=1 → Score = 2
    • Total score: 4
Simulation For Team Shellphish

- R03: Competition fields CB1, RB2, CB3
  - Points for round 02
    - CB1: Availability=1, Security=1, Evaluation=1+(3/6) → Score = 1.5
    - RB2: Availability=0.8, Security=2, Evaluation=1 → Score = 1.6
    - CB3: Availability=1, Security=2, Evaluation=1 → Score = 2
    - Total score: 5.1