Sys: A static/symbolic tool for finding good bugs in good (browser) code

Fraser Brown, Dawson Engler, Deian Stefan
Goal: automatically find security bugs in browsers
Goal: automatically find security bugs in browsers

“Problem” 1: Browsers check a lot
ClusterFuzz provides many features to seamlessly integrate fuzzing into a software project’s development process:

- Highly scalable. Google’s internal instance runs on over 25,000 machines.
Santizers (detect errors as program executes)

Clang 12 documentation

ADDRESSSSANITIZER
Static checkers (look for “buggy patterns” in source code)

Coverity Scan: Firefox

<table>
<thead>
<tr>
<th>Project Name</th>
<th>Firefox</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of code analyzed</td>
<td>8,149,652</td>
</tr>
<tr>
<td>On Coverity Scan since</td>
<td>Feb 22, 2006</td>
</tr>
<tr>
<td>Last build analyzed</td>
<td>9 days ago</td>
</tr>
</tbody>
</table>
Static checkers (look for “buggy patterns” in source code)

Mach static analysis
It is supported on all Firefox built platforms. During the first run it automatically installs all of its dependencies like clang-tidy executable in the .mozbuild folder thus making it very easy to use. The resources that are used are provided by toolchain artifacts clang-tidy target.

This is used through `mach static-analysis` command that has the following parameters:

- `--check`: Runs the checks using the installed helper tool from ~/.mozbuild.
- `--checks, c`: Checks to enable during the scan. The checks enabled in the yaml file are used by default.
- `--fix, f`: Try to autofix errors detected by the checkers. Depending on the checker, this option might not do anything. The list of checkers with autofix can be found on the clang-tidy website.
- `--header-filter, h-f`: Regular expression matching the names of the headers to output diagnostic from. Diagnostic from the main file of each translation unit are always displayed.

As an example we run static-analysis through mach on dom/presentation/Presentation.cpp with `google-readability-braces-around-statements` check and autofix we would have:

```
./mach static-analysis check --checks=".*", google-readability-braces-around-statements" --fix dom/presentation/Presentation.cpp
```

If you want to use a custom clang-tidy binary this can be done by using the `Install` subcommand of `mach static-analysis`, but please note that the archive that is going to be used must be compatible with the directory structure clang-tidy from toolchain artifacts.

```
./mach static-analysis install clang.tar.gz
```
Static checkers (look for “buggy patterns” in source code)

**Mach static analysis**

It is supported on all Firefox built platforms. During the first run it automatically installs all of its dependencies like clang-tidy executable in the .mozbuild folder thus making it very easy to use. The resources that are used are provided by toolchain artifacts clang-tidy target.

This is used through `mach static-analysis` command that has the following parameters:

- `-v` - Verbose.
- `-r` - Run all checks.
- `-check` - Runs the checks using the installed helper tool from --mozbuild.
- `-checks` - Checks to enabled during the scan. The checks enabled in the vami file are used by default.

---

**Static Analysis Bounty**

In coordination with the GitHub Security Lab, we have launched a new program that rewards the submission of static analysis tools that identify present or historical security vulnerabilities in Firefox. We will accept static analysis queries written in CodeQL or as clang-based checkers (clang analyzer, clang plugin using the AST API or clang-tidy). Submissions should be made following our instructions below.
Bounty programs (pay $$$$ for bug reports)
Client Bug Bounty Program

Introduction

Chrome Vulnerability Reward Program Rules

The Chrome Vulnerability Reward Program was launched in January 2010 to help reward the contributions of security researchers who invest their time and effort in helping us to make Chrome and Chrome OS more secure. Through this program we provide monetary awards and public recognition for vulnerabilities responsibly disclosed to the Chrome project.
Bounty programs (pay $$$ for bug reports)

Pwn2Own Researchers Exploit Mozilla Firefox, Microsoft Edge and Tesla

By: Sean Michael Kerner | March 22, 2019
Bounty programs (pay $$$ for bug reports)

Pwn2Own Researchers Exploit Mozilla Firefox, Microsoft Edge and Tesla

By: Sean Michael Kerner | March 22, 2019

Pwn2Own 2019: Hackers can now scoop $80,000 for Chrome exploits

James Walker 16 January 2019 at 15:52 UTC
Updated: 26 November 2019 at 11:09 UTC
Goal: automatically find security bugs in browsers
Goal: automatically find security bugs in browsers

Problem 2: Static checking didn’t find much
Goal: automatically find security bugs in browsers

Coverity Scan: Firefox

<table>
<thead>
<tr>
<th>Project Name</th>
<th>Firefox</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of code analyzed</td>
<td>8,149,652</td>
</tr>
<tr>
<td>On Coverity Scan since</td>
<td>Feb 22, 2006</td>
</tr>
<tr>
<td>Last build analyzed</td>
<td>9 days ago</td>
</tr>
</tbody>
</table>

Last sec-critical and sec-high bugs: 2014

(thanks Edward Chen!)
Goal: automatically find security bugs in browsers

Problem 3: Symbolic execution is hard and slow
New approach: Static checking + underconstrained symbolic execution
New approach:
Static checking + underconstrained symbolic execution

Look for “buggy patterns”
New approach: Static checking + underconstrained symbolic execution

“Run” program over all possible values
New approach:
Static checking + underconstrained symbolic execution

Start anywhere
New approach: Static checking + underconstrained symbolic execution
New approach: Static checking + underconstrained symbolic execution

- Static analysis identifies many potential errorsites ($$)
New approach: Static checking + underconstrained symbolic execution

- Static analysis identifies many potential errorsites ($)

- Symbolic execution jumps directly to candidate errorsite and executes ($$$$$)
New approach: Static checking + underconstrained symbolic execution

- Static analysis identifies many potential errorsites ($)

- Symbolic execution jumps directly to candidate errorsite and executes ($$$)$
New approach: Static checking + underconstrained symbolic execution

- Static analysis identifies many potential errorsites ($)
  - Programmer-written static extension (max 273 LOC)
- Symbolic execution jumps directly to candidate error site and executes ($$$$$)
New approach: Static checking + underconstrained symbolic execution

- Static analysis identifies many potential errorsites ($)
  - Programmer-written static extension (max 273 LOC)
- Symbolic execution jumps directly to candidate errorsite and executes ($$$$$)
  - Programmer-written symbolic checkers (max 106 LOC)
New approach: Static checking + UC symbolic execution
New approach: Static checking + UC symbolic execution

: ModuleID = 'undef.bc'
source_filename = "undef.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

LLVM IR File(s)
New approach: Static checking + UC symbolic execution

: ModuleID = 'undef.bc'
source_filename = "undef.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

LLVM IR File(s)

Static extension

Alloca x => Uninit x
Store y x => Init x
Load x => Error x
...

Allocated x => Uninit x
Store y x => Init x
Load x => Error x
New approach: Static checking + UC symbolic execution

LLVM IR File(s)  

Static extension

Suspicious path

: ModuleID = 'undef.bc'
source_filename = "undef.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

Alloca x => Uninit x
Store y x => Init x
Load x => Error x

Suspicious path
New approach: Static checking + UC symbolic execution

LLVM IR File(s) → Static extension → Suspicious path → Symbolic checker

- ModuleID = ‘undef.bc’
- source_filename = "undef.c"
- target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
- target triple = "x86_64-pc-linux-gnu"

Alloca x => Uninit x
Store y x => Init x
Load x => Error x
...

Alloca x
Store y z
Load x

V = Load shadow x
If isSet V
Then Bug
Else No Bug

Suspicious path

Symbolic checker
New approach: Static checking + UC symbolic execution

ModuleID = 'undef.bc'
source_filename = "undef.c"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

LLVM IR File(s)

Static extension

Suspicious path

Symbolic checker

Alloca x => Uninit x
Store y x => Init x
Load x => Error x

V = Load shadow x
If isSet V
Then Bug
Else No Bug

Bug
Walk through heap out-of-bounds bug, CVE 2019-5827
Static extension (heap out-of-bounds)
Static extension (heap out-of-bounds)

```c
a = sqlite3_malloc( (sizeof(u32)+10)*nStat );

memset(a, 0, sizeof(u32)*(nStat) );
```
Static extension (heap out-of-bounds)

```c
a = sqlite3_malloc( (sizeof(u32)+10)*nStat );

memset(a, 0, sizeof(u32)*(nStat) );
```
Static extension (heap out-of-bounds)

\[
a = \text{sqlite3}_3\_\text{malloc}( (\text{sizeof(u32)}+10)\*\text{nStat} ) ; \\
\text{memset}(a, 0, \text{sizeof(u32)}\*(\text{nStat}) ) ;
\]
Static extension (heap out-of-bounds)

```c
a = sqlite3_malloc( (sizeof(u32)+10)*nStat );

memset(a, 0, sizeof(u32)*(nStat) );
```
Static extension (heap out-of-bounds)

\[ a = \text{sqlite3\_malloc}( (\text{sizeof}(\text{u32})+10) \times \text{nStat} ) ; \]

\[ \text{memset}(a, 0, \text{sizeof}(\text{u32}) \times (\text{nStat}) ; \]
Static extension (heap out-of-bounds)

```c
a = sqlite3_malloc( (sizeof(u32)+10)*nStat );

memset(a, 0, sizeof(u32)*(nStat) );
```
Symbolic checker (heap out-of-bounds)

\[a = \text{sqlite3\_malloc}\left((\text{sizeof}(\text{u32})+10)\ast\text{nStat}\right);\]

\[\text{memset}(a, 0, \text{sizeof}(\text{u32})\ast(\text{nStat}) );\]
Symbolic checker (heap out-of-bounds)

\[ a = \text{sqlite3\_malloc}( (\text{sizeof}(\text{u32})+10)\times n\text{Stat} ); \]

\[ \text{memset}(a, 0, \text{sizeof}(\text{u32})\times(n\text{Stat})) ; \]

\[ \text{sizeof}(\text{u32}) \times n\text{Stat} \geq (\text{sizeof}(\text{u32}) + 10) \times n\text{Stat} \]
“Constraints” express lines of code as logical formulas

\[ a \& b \& c | d | e \ldots \]
“Constraints” express lines of code as logical formulas

\[ a \land b \land c \lor d \lor e \ldots \]

\[ a = true \\
\[ b = true \\
\[ c = true \\
\[ d = false \\
\[ e = true \]
“Constraints” express lines of code as logical formulas

a & not a & b & c | d | e ....
“Constraints” express lines of code as logical formulas

\( a \& \neg a \& b \& c \mid d \mid e \ldots \)

Unsat
“Constraints” express lines of code as logical formulas
“Constraints” express lines of code as logical formulas

```
malloc (y)
x [y -1]
```

Suspicious path
“Constraints” express lines of code as logical formulas

```
... 
malloc (y) 
x [y -1] 
```

Suspicious path

```
x = 0xdeadbeef 
tmp = y - 1
... 
```

Constraints
“Constraints” express lines of code as logical formulas

```
...  
malloc (y)  
x [y -1]  

x = 0xdeadbeef  
tmp = y - 1  
...
```

Suspicious path

```
x [y -1]  
```

Constraints

```
y - 1 > y
```

Bug constraints
“Constraints” express lines of code as logical formulas

Suspicious path

Constraints

Bug constraints

SMT Solver
Symbolic checker (heap out-of-bounds)

```c
a = sqlite3_malloc( (sizeof(u32)+10)*nStat );
memset(a, 0, sizeof(u32)*(nStat) );
```
1. Symbolic engine translates line

\[ a = \text{sqlite3_malloc}\left( (\text{sizeof}(\text{u32})+10)\times\text{nStat} \right); \]

\[ \text{memset}(a, 0, \text{sizeof}(\text{u32})\times(\text{nStat})); \]
1. Symbolic engine translates line

```c
a = sqlite3_malloc( (sizeof(u32)+10)*nStat );

memset(a, 0, sizeof(u32)*(nStat) );
```
Symbolic checker (heap out-of-bounds)

```c
a = sqlite3_malloc( (sizeof(u32)+10)*nStat );

memset(a, 0, sizeof(u32)*(nStat) );
```
2. Symbolic checker asks for OOB

\[ a = \text{sqlite3\_malloc}( (\text{sizeof(u32)+10}) \times \text{nStat} ); \]

\[ \text{memset}(a, 0, \text{sizeof(u32)} \times (\text{nStat}) ); \]

\[ \text{sizeof(u32)} \times \text{nStat} \geq (\text{sizeof(u32)} + 10) \times \text{nStat} \]
3. Query SMT solver
Results:

- 4 checkers (2 out-of-bounds, 1 uninitialized memory, 1 UAF)
- 51 bugs (43 confirmed), 18 false positives
- 3 browser bug bounties (17 total bugs)
- 4 browser CVEs (18 total bugs)
- 2 browser security audits
- One Coverity re-configuration

mlfbrown@stanford.edu