Tavis Ormandy and Will Drewry
Google, Inc.
January, 2008

Regular Exceptions
IT-DEFENSE 2008

Tavis Ormandy and Will Drewry
Google, Inc.
January, 2008

Introductions

Who are we?

- Information Security Engineers at Google
- UNIX and open source security researchers

Why are we here?

- Complex code is an appealing attack surface
- Some very complex code is taken for granted

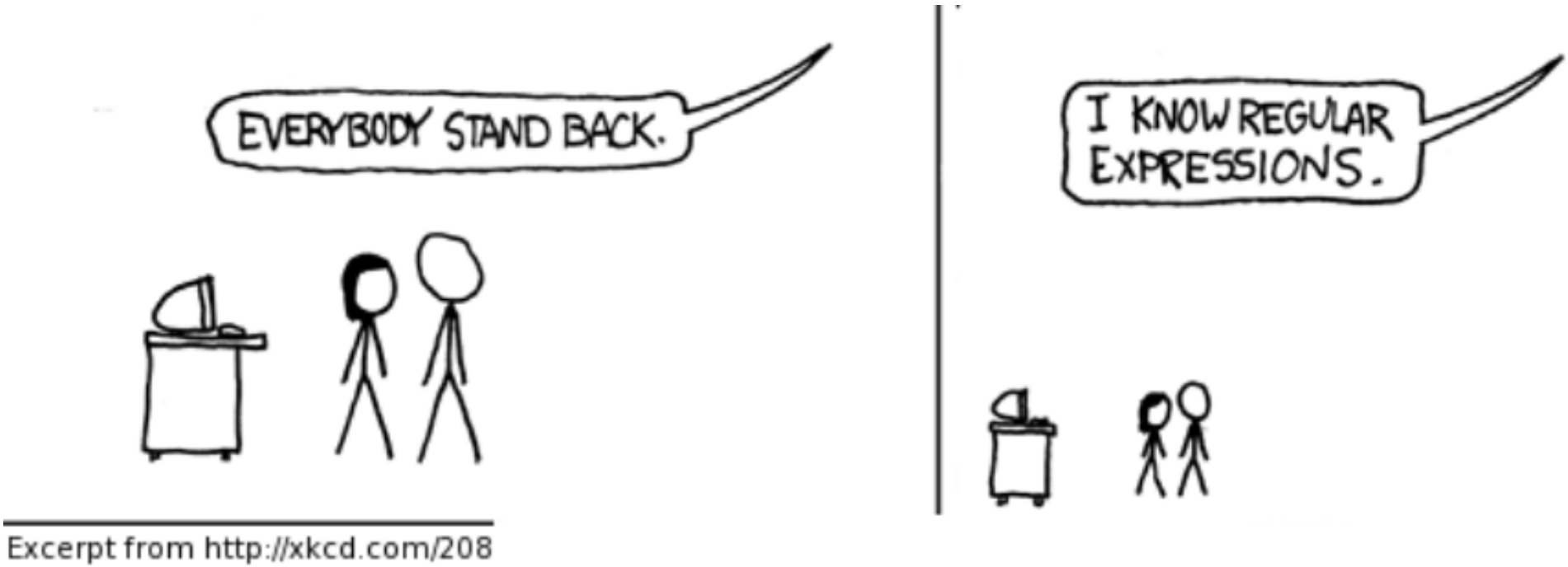
Why We're Here

A few weeks of research; major software broken

- Remote code execution:
  - Adobe Flash Player, Adobe Acrobat, Apple Safari, Konqueror, and many more
- Denial of service attacks against Tcl, PostgreSQL, Python, and others
- Abuse the OpenPGP web of trust to compromise GnuPG
- Compromise the interpreters of multiple popular scripting languages

Google

How?



Excerpt from http://xkcd.com/208

Google

Regular Expressions?

Regular expressions are everywhere

- Your browser, blog, programming language, database, mail server, mobile phone all provide some facility for regular expressions.
- Regular expressions are extremely complex and still an active research topic.
- They're not just hard but NP-Hard!

History

We're not the first to attack regular expressions

- iPhone hack with Safari's JavaScript Regexp() object
- Some regular expressions are known to kill Cisco IOS.
- Well-known practical joke:

```
$ egrep 'A{200}{200}{200}' /etc/passwd
```

- However, no complete “Regex Security Handbook” exists.

Problems

Regular Expressions are difficult

- Generally handled in multiple passes, building up state machines or executable bytecode
- Solving regular expressions is hard work
  - Large Input, Large Patterns, Submatch Extraction, Back references
  - etc.

Google

Attacking Regular Expressions

Audited multiple engines manually and automatically

- Designed an automated testing procedure
- Researched regular expression theory from a security perspective
- Audited multiple popular implementations
- Found multiple exploitable flaws
- Documented various “general” attacks

Regex Fuzzing

- Documented/Fingerprinted cross-engine features
- Randomly generate from known seeds
- Construct from the inside out
  - [:alpha:] => [^[:alpha:]] => ?([^[:alpha:]]) ...
- Perform the tests as quickly as possible
  - Native language construction: c, javascript, actionscript and SWIG
- Tried EBNF-based fuzzer
  - Tedious to do language ports
  - Structural biases too annoying

(Typical) Regular Expression Execution

Three stages:

- **Parse.**
  - Initial pass over expression
- **Compile.**
  - Expression converted into internal representation
- **Execute.**
  - Construct state machine attempting match.

Google

Parsing

Regular expression matching begins with parsing

- An initial pass is sometimes required to estimate space requirements
- Determine “class” of expression if multiple types supported
- Simple correctness checks (balanced parentheses, recognised named classes, etc.
- Some engines parse and compile in one pass.

Parsing Attacks

We found several parser attacks

- Cause mismatches between passes
- Make memory estimates wrong
- Make the parser seek too far (\c\)
- Trick the parser to not recognise constructs (E.g., hide one expression inside another).

Memory Estimation Errors

- Not all engines do parse-based estimation
- Estimation ...
  - **can** be inaccurate
  - **must** handle worst-case scenarios
- Not too hard to confuse the estimation logic:
  - Poorly handled wide characters
  - Error in the bytecode counter

Memory Estimation Errors: Impact & Mitigation

- Can result in buffer overflow or invalid bytecode (PCRE)
- Mitigation tricky due to checking complexity

Compile

Conversion into an internal representation

- Several implementations use a bytecode representation that executes in a pseudo-vm
- Historically, compiling to machine code was common
- Compilation often includes optimizations

Compile Attacks

We identified multiple vulnerabilities in compilation

- Poorly handled allowed expressions
- Poor evaluation of logically impossible expressions
- Worst-case compilation time

Google

Pattern Decoding/Parsing 2nd Pass

- Poorly parsed backreferences
- Malformed unicode
- Poorly handled control characters
- Poorly handled quantifiers or intervals

Pattern Decoding: Impact

- Controllable heap overflows due to modal desynchronization in Perl and PCRE [demos]
- Integer overflows via poor backreference parsing E.g., PCRE [demo]
- Heap corruption due to uncapped intervals lacking verifiable internal representation in ICU [demo]
- ...

Google

Pattern Decoding: Mitigation

- Detecting bad expressions is as difficult as implement the regular expression itself
- Possible solutions include:
  - whitelisting known values
  - only supporting pre-validated regular expressions
  - don't let users insert regex instructions (regex injection! See GnuPG)
  - ...

Google

Impossible Expressions

- Regular expressions allow for patterns that evaluate to nothing
- These are often logical invalid but are represented in code poorly

Impossible Expressions: Impact

- Usually, these issues result in NULL valued nodes or other aberrant behavior
- Out of bounds reads and NULL dereferences in Boost:: Regex++ [demo]
- NULL dereference in PCRE unicode properties
- Our own internal library had trouble with empty character classes: [^\s\S]
- GNU's egrep mishandled empty, quantified submatches: () {1024}{1024}{1024}
- ICU's backreferences used an assert() to ensure it evaluated to 1-10 which was compiled out [demo]
- OOB read on > INT_MAX back references in PostgreSQL/Tcl [demo]

Impossible Expressions: Mitigation

- Use updated regular expression libraries
- Do not accept untested patterns
- Compile code without -DNDEBUG
  - Experience suggests developers seldom use assert() correctly.
- You'd have to preparse the value to know which layers on more potential for vulnerable code.

Execution

Execution exposes many undesirable characteristics

- An attacker will want to expose worst-case behaviour, potentially resulting in
  - Exponential Resource Usage
  - Crashes
  - Infinite Loops
  - Etc.

Google

Stack overflows

- Recursion is a common design pattern
- Backtracking is one example
  - Common mechanism for exploring possible matches . . .
  - With leftmost longest match . . .
  - With backreferences
- Pathological expressions lead to stack overflows
- Some expressions can be forced to backtrack

Stack overflows: Impact

- Arbitrary expressions, or even input, may lead to unexpected termination
- The majority of NFA-based engines suffered from this
- PCRE is one major example.

```
$ echo '=XX================================'  |
    pcregrep 'X(.+)+X'
```

Stack overflows: Mitigation

- Compile-time or run-time recursion bounds
  - PCRE already has this as do others
  - Set it to a reasonable limit for your needs
- Craft regular expressions carefully
  - Embed an upper limit in the pattern itself:
    - (.?)* may be rewritten as (.?){,1024}

Memory Exhaustion

- Majority of engines require asymmetric memory consumption
- Any engine with backtracking support
- Several allow small input expressions to consume orders of magnitude
- May be due to internal representation:
  - a{5} may be represented as aaaaa
  - a{500000} ....
- There are well-known examples: (man grep)

  grep -E 'a{200}{200}{200}' /etc/passwd

Memory exhaustion: Impact

- May result in easily handled errors or worse
- OOM killer
- Unchecked/dereferenced NULLs (StrongARM? Barnaby?)
- Examples: GNU regcomp, GNU grep, postgresql/tcl duptraverse(), perl memoization, ...

Memory exhaustion: Mitigation

- Memory-bound the process or the library (if supported)
- Write regular expressions with this risk in mind
  - Well written static expressions can eliminate input-only attacks.
- Use DFA-only engines if backreferences aren't needed

Google

Exponential Computation

- Many engines use NFAs or NFA-to-DFA construction
- Supporting back references and submatch extraction often leads to edge cases with exponential computational requirements
- All possible states may be computed during computation
- Alternatively, on-demand
- May require years to run
- In addition, sometimes state optimizations lead to infinite loops

Exponential Computation: Impact

- Affects nearly every regular expression implementation
- Denial of service attacks are the primary risk
- Can take out backend databases, etc (postgresql, ...)
- May affect other services running on the same host

Google

Exponential Computation: Mitigation

- Some engines support enforced maximums:
  - Number of states (TCL, lex, ...)
  - Total run time
- Whitelisted regular expressions (as in GnuPG)

Higher-level impact (1/2)

- Each of the engine deficiencies discussed do not directly result in a security risk
- It's only when exposed to hostile users it becomes a problem.

Higher-level impact (2/2)

- PDF standard supports javascript by **default**
  Acroread -> Old javascript engine -> rxspencer
- OpenPGP **trust** signatures: GnuPG -> glibc
- Web Browser -> JS/Actionscript -> PCRE/etc:
  Safari -> JS -> PCRE
  * -> **Flash** -> PCRE
- Webapp -> Scripting engine/Database/indexer
  - LXR -> swish/glimpse -> PCRE/GNU libc
  - Rails -> ActiveRecord Extensions -> ...

Guidelines for safety

- Think before exposing regular expressions
- Time and memory bound all phases if possible
- Limit recursion and/or backtracking (pcre, ...)
- Limit number of NFA states:
  - javascript.options.relimit in Firefox, compile-define in tcl/postgresql
- Only use whitelisted pattern
- Craft patterns to avoid exponential evaluation
- Run in a subordinate process
  - Possibly even jailed (systrace, etc)

Conclusion

- Our tests were simplistic; our results were terrifying
- Just the tip of the iceberg
- Consider carefully when exposing complex code
- A regular exception doesn't make it a correct one

What now?

- Apply mitigation tactics from above
- Integrate regular expression testing in to your regular tests
  - Our fuzzing code might be a good start!

Questions?

Project page: code.google.com/p/regfuzz
Release date: February 15, 2008