Robert J. Hansen
Meredith L. Patterson

BLACK HAT BRIEFINGS

Stopping Injection Attacks with Computational Theory

Input validation is an important part of security, but it's also one of the most annoying parts. False positives and false negatives force us to choose between convenience and security—but do we have to make that choice? Can't we have both? In this talk two University of Iowa researchers will present new methods of input validation which hold promise to give us both convenience _and_ security. A basic understanding of SQL and regular expressions is required.

Robert J. Hansen: *B.A. in Computer Science from Cornell College, 1998. Graduate student at the University of Iowa, 2003-2005, researching secure voting systems with Prof. Doug Jones. Senior Security Engineer at Exemplary Technologies, 2000; Cryptographic Engineer at PGP Security, 2000-2001.*

Meredith L. Patterson: *B.A. English (Linguistics) from the University of Houston, 2000. M.A. Linguistics from the University of Iowa, 2003. Graduate student at the University of Iowa, 2003-2005, studying data mining with Prof. Hwanjo Yu. Bioinformatics intern at Integrated DNA Technologies, 2003-2005.*

Stopping Injection Attacks with Computational Theory

Robert J. Hansen
Meredith L. Patterson
The University of Iowa

Part I: What's the Problem?

(Short Answer: Lots!)

What's the Problem?

- Users input garbage
  - Garbage in, garbage out…
  - Attackers input garbage, too
  - Garbage in, exploits out.
- We don't have good tools to keep garbage out
  - Current IV techniques are largely ineffective.
  - Most Web developers don't bother—is this because of laziness or because there are no good tools?

The State of the Art

- Common wisdom: "use regexps to validate user input"
  - False positives: some legitimate inputs will be incorrectly flagged as bad, leading to user frustration
  - False negatives: some attacks will be incorrectly flagged as safe, leading to exploits
- Which is better? To err on the side of convenience, or err on the side of security?

The Nonsense Of It All

Either way, we're *choosing to make errors.*

Why are we choosing to make errors?

Why does conventional wisdom encourage a one-or-the-other approach?

Shouldn't we try to find techniques which reduce the error rate?

The Real World

"Be careful when making [data validators] more restrictive, though, because a [validator] that rejects one percent of valid input is far more annoying than one that lets through ten percent of invalid data."

Luke Welling and Laura Thomson,
PHP and MySQL Web Development.
Sams Publishing, 2005

Lots of Books Are That Bad

- Welling and Thomson weren't unusual.
- In our survey of commonly available Web development books, only two discussed validation in any detail.
- If that could get through technical review process, that means either the review was braindead or our best-practices are.
- How many of us have been led to believe we can have convenience or security, but not both?

Convenience AND Security

- Convenience *must* be secure, otherwise we're condemned to 0-day and script kiddies.
- Security *must* be convenient, otherwise nobody will use secure systems.
- False positives and negatives are *categorically unacceptable* and we must seek more accurate technologies.
- Our belief that we can only have one or the other is *crippling* web security.
- We have to change the way we think. We have to demand convenience *and* security.

And that's what we're offering.

We've discovered a way to apply theoretical computer science

(aka "the math weenie stuff we thought we'd never use")

… to giving us near-100% protection against injection attacks, and

… near-100% protection against false alarms.

(When used properly.)

It won't solve all security problems and it won't protect you from faulty implementations.

But it's a start, and it's a lot better than everything else we have so far.

Let's Talk Problem.

Anywhere you have user input, you have input validation problems

Injection attacks are a specific kind of input validation problem

- You want to allow users to change the flavor of a command string…
- … but not the functionality of a command string

Has anyone tried to solve it?

- Beizer (1983) considered the problem intractable, which killed all future research.
  - Validating user inputs is AI-complete. Get perfect input validation and you get strong artificial intelligence.
  - Input validation is a huge problem space. Most of it is in the land of Mordor, and you don't want to go there, Mr. Frodo.
  - But there are islands of tractability, and we've discovered some of them are *really cool.*

A Simple Injection Example

- SQL: "SELECT RIGHTS FROM ACCESS_TABLE WHERE PASSWORD = '$foo'"
  - Works great if $foo is "squeamish ossifrage"
  - Not so great if $foo is "' OR '1' = '1"
  - How can we differentiate injection attacks from legitimate inputs?

Part II: Theory

Or, "The Math Geekery We All Hated In Our First Semester of Graduate School"

Syntax versus Semantics

- Syntax: the structure and order of a sentence
- Semantics: the meaning of a sentence
- Legitimate inputs only change semantics
  - If $foo is "squeamish ossifrage", the resulting SQL statement has a different meaning than if it was "Hi, Black Hat!"
- Injections change syntax *and* semantics
  - All injection attacks work by adding additional elements to a control statement
  - These necessarily result in syntactic changes

Computational Theory

- Concerned with syntax and semantics
  - We encode math problems as strings
  - … and use regexp-like devices to compute on those strings
- Concerned with the limits of computers
  - If CT says a problem can't be solved, you won't get a computer to solve it.
- So let's look at regexps from a CT perspective.
  - Can regexps handle complex syntax?
  - If not, why are we doing validation with them?

Finite State Automata

- Very simple model of computation
- Take a string of letters, and based on each letter, move to a new state
- If at the end of the input you're in an "accepting" state, the string is a good input
- Otherwise, it's bad.
- It has no memory, no recursion, no anything: all it knows is its current state and the next symbol it's looking at.

FSA II

FSA are equivalent to regexps; anything a regexp can do, an FSA can do

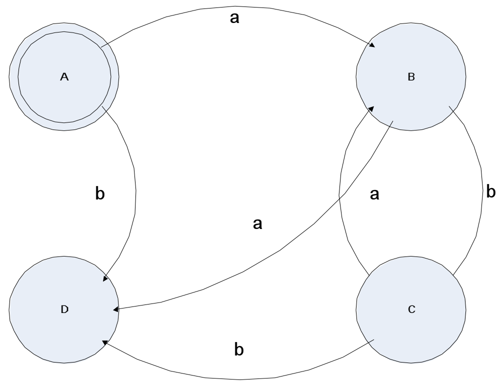
… and everything an FSA can't do, a regexp can't do.

Perl regexps are a little different; we'll cover them in a bit. But that said, on with the show.

FSA III: Example

- Our language is made up of "a" and "b"
  - We'll never see anything that's not in our language
  - In the Real World, this would be ASCII or Unicode.
- We want to accept "(ab)*"
  - "", "ab", "abab", "ababab"… etc.
- In our next slide, states A and C accept

FSA IV: The State Diagram



If our input ends when we're in States A or C, our FSA matches the regexp (ab)*. Anything else, and we're out of luck.

Note that this works for *any* length sequence of as and bs, despite our limited number of states. At some point we'll recycle a state.

Summation of FSAs

- All regular expressions can be described by finite-state automata (FSA).
- FSA have (drumroll, please) a limited number of states.
- For a large enough input, you're going to revisit a state.
- Just like if you're in a theater with 100 seats, if there are 101 tickets sold, a couple of people are going to get real friendly.

The Limits of FSAs

- Let’s imagine we have an FSA which will recognize $a^k b^k$.
  - (That is, any number of as followed by the same number of bs.)
  - Can this FSA exist? Will it be reliable?
- If it can’t exist or isn’t reliable, where does that leave us?
  - (Answer: in the land of Mordor, where Mr. Frodo doesn’t want to go.)

The Pumping Lemma

- If we have a long enough input, we have to repeat (“pump”) some portion of our FSA.
  - This is a consequence of our FSA being… well… F.
- This turns out to be its downfall.
- If after *k* repetitions of “a”, we’re in a state where we can recognize *k* repetitions of “b”…
- … then can we revisit that same state after *k+m* repetitions of “a”, and go on to recognize *k* repetitions of “b”?

Oops, We Did It Again.

- We've just found a language which looks like we should be able to recognize it with a regex, but…
- … *no regex can ever recognize it.*
- Not reliably, at least.
  - It'll have a lot of false positives and false negatives.
  - We'll be forced to choose between convenience and accuracy.
  - Sounds strangely like where we currently stand, doesn't it?
- So if regexs aren't strong enough to recognize that language, what is?

Context Free Grammars

- A *context free grammar* is the next most powerful kind of computer.
  - (For those who care, Turing Machines come after this.)
- A CFG is basically a regexp which is allowed to recurse. Kind of like a Perl regexp, in other words.
- Perl doesn't have regexps. It has CFGs.

An Example

CFGs generate strings of letters according to rules. Let's come up with a rule that will recognize $a^k b^k$.

Our Rule A says:

- Generate aAb
- Or generate nothing
- (where "A" means "another repetition of Rule A")

Does this get us anything?

Let's See

(empty) = A   nothing

ab = A   aAb, A   nothing

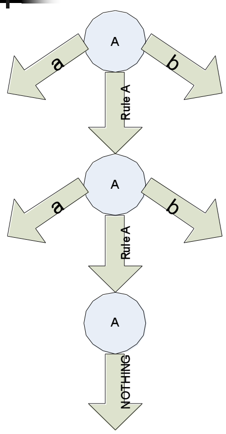
aabb = A   aAb, A   aAb, A   nothing

aaabbb = A   aAb, A   aAb, A   aAb, A   nothing

… by Jove, I think we've got it!

We can recognize this language!

Derivation Tree



Rule A comes in two varieties: one which generates three outputs (an "a", then another application of Rule A, then a "b"), and one which generates nothing at all. If we want to generate the string "aabb", this is the derivation tree we'd use. Read counterclockwise from the upper left, reading only the symbols in gray arrows which lead nowhere.

If we wanted to make sure "aabb" was in our language—validation instead of generation—we'd just see if we could generate "aabb" from our set of rules. From a math standpoint, generating strings *is* validating strings.

YES!

We've found something stronger than a regular expression.

If we needed user inputs that matched $a^k b^k$, we literally couldn't do it with a regular expression.

We needed a context-free grammar instead.

This led us out of Mordor.

This leads us to our first axiom of input validation:

Minimally Strong Mechanism

Input validation needs to be done with a mechanism strong enough to recognize the language

SQL, for instance, is a context-free language…

... as is HTTP, as are command shells, as are programming languages, as is…

… so we mustn't validate them using regexps!

Summary

HTTP and SQL are both context-free languages, and need context-free grammars to validate them.

We've seen regular expressions cannot validate context-free languages.

But the hacker word-of-mouth tells people to use regular expressions to validate user inputs to SQL and Web servers.

We are living in sin.

Recommendations

- Don't validate naked user inputs
  - Always validate it in the context of how it'll be used—as a complete SQL statement, as a complete HTTP request, etc.
- Use the right tool for the job
  - SQL is a context-free language, so validate using a context-free grammar
  - Weaker tools won't do the job
- Don't get stronger than you have to
  - User inputs will be attacked. Lots.
  - If you get compromised, you want to give your attacker the *weakest* resources possible
  - So don't use stronger mechanisms than you really have to

Recommendations, Part Two

- Learn basic computational theory
  - Art Fleck's *Formal Models of Computation*
  - Michael Sipser's *Introduction to the Theory of Computation*
- Read our academic paper: "Guns and Butter: Towards Formal Axioms of Input Validation"
  - Should be in your media package
  - Available on the Web at http://cs.uiowa.edu/~rjhansen/HP2005.pdf
- Stop thinking of validation in terms of convenience or security
- Start thinking of validation in terms of convenience *and* security

From Theory into Practice

Dejection Has Never Felt So Good

Injection Attacks: A Refresher

- Solving input validation is AI-complete, because of the semantics problem.
  - Semantics means "meaning". If you could make a program that could recognize the meaning of things and flag bad stuff, you'd have a program that could teach you philosophy, ethics and morality.
- But injection has a *syntactic* component as well as a *semantic* component
- Injection is a special case of input validation, and it is solvable.

Secure Sublanguages

Imagine this SQL statement:

SELECT RIGHTS FROM ACCESS_TABLE
WHERE NAME="[string]" AND
PASSWORD="[string]"

That's valid SQL. An injection attack would make it look different.

So let's make a sublanguage of SQL, where our statement is the *only* kind of statement we can make.

Secure Sublanguages II

So SELECT RIGHTS FROM ACCESS_TABLE WHERE NAME="root" AND PASSWORD="" OR "1"="1" would…

Not…

… be a valid statement in our sublanguage. We've added syntactic and semantic elements.

We can detect the extra syntactic elements using a CFG.

Secure Sublanguages III

[This slide will have a monstrously huge SQL derivation tree showing the first statement]

Secure Sublanguages IV

[This slide will have a truly grotesque derivation tree showing the second statement]

That's… Obnoxious.

- But the differences between the two are obvious.
- A CFG will be able to easily spot the difference between the two.
- What we need is a known-good parse to compare the user input against.
- If the user input has an identical parse tree, we know the input hasn't been injected.

How Can This Be Automated?

- Web developers can create an *exemplar string,* which is an example of what inputs should look like.
  - They can then mark portions of the exemplar string as *mutable*
  - For instance, the password field in a WHERE PASSWORD='foo' clause is (usually) mutable
- That exemplar string is pre-parsed, and an XML derivation tree generated
- From there, whatever input the user gives is fed into our tool and its own XML derivation tree is generated
- We compare the derivation trees: any differences outside the mutable sections is an additional syntactic element—and thus an injection attempt!

Dejection

We call this counter-injection technique *dejection,* both as a pun on "injection" and because we hope to make script kiddies very dejected.

Dejection can be applied to any SQL database for which we can get a yacc file

- Oracle, DB2, SQL Server, SQLite, MySQL…
- … and PostgreSQL is already done (libdejector-pg).

So How Does It Work?

That varies according to the language.

libdejector-pg is a C library with SWIG wrappers

- Python, Perl, PHP, Ruby, Java, .NET…
- … if SWIG supports it, we can build it.

So let's look at a hypothetical example. The real Python bindings might look a little different, but they'll be close.

libdejector-pg-python

```
exemplar = Dejector.MakeExemplar
    ("SELECT RIGHTS FROM
ACCESS_TABLE WHERE  NAME='[eli]'
    AND PASSWORD='[b14ckh47]'")
if exemplar.Validate(userInput):
    executeSQLQuery(userInput)
else:
    sendScriptKiddieAway()
```

That's Simple!

- It's designed to be.
- You create an exemplar, you validate user input against it, you're done.
- With a properly-written exemplar, you have significant resistance to SQL injection attacks and significantly reduced false positives and negatives.
- In theory, it's 100% accurate. But anything that works 100% in theory never works 100% in practice, so let's not get too carried away.
- It's a tool. It's a *good* tool. Please use it.

Licensing

- Yes, we're looking into software patents.
  - Are they evil? Yes. Do we have student debts to pay off? Lots.
  - Any project released under an OSI-approved license will receive royalty-free licensing.
  - Proprietary projects will need to talk to us.
- libdejector-pg is released under GNU GPL.
- If you're interested in dejection, please talk to us!

Contact Information

Rob Hansen and/or Meredith Patterson
Department of Computer Science
The University of Iowa
Iowa City, Iowa 52240
rjhansen@cs.uiowa.edu
mlpatter@cs.uiowa.edu

Questions and Answers