A dynamic technique for enhancing the security and privacy of web applications

Ariel Futoransky, Ezequiel Gutesman and Ariel Waissbein
Core Security Technologies

August 2, 2007 - Blackhat Briefings

Outline

1. Introduction
2. Known Tools
3. Classical Taint Mode
4. Classical Taint mode and Character Granularity Information
5. Introducing CORE GRASP
6. Future Work

Introduction

1. Why worrying about injection attacks?
2. Motivating our work. Objectives, Results.
3. Demo.
4. Describing threats.
5. Known countermeasures.

Why worrying about injection attacks?

- Web application vulnerabilities are discovered every day. Most of the exploits make use (or abuse) of injection vulnerabilities.
- Exploitation of these vulnerabilities leads to numerous problems:
  - Data theft / alteration.
  - Impersonation, private data disclosure.
  - Remote code execution (in client's browsers).
- As programming languages evolve and lower the learning curve for new developers they fail in introducing protection mechanisms to prevent these attacks.
- Privacy can be compromised. User private data (such as credit card information) can be stolen. User credential theft can lead to impersonation.

Objectives

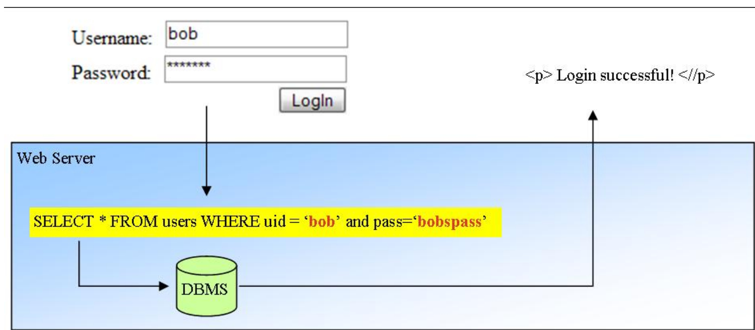
Exhibit a technique that can be applied to **any** web language with the following properties:

- It protects all webApps, without requiring further changes to its source code or the network architecture.
- Injection attacks are detected and may be (optionally) blocked.
- It allows a site owner to enforce privacy policies over the data managed by the WebApp.
- Notice that if injection attacks and privacy violations are blocked the system should be very accurate!

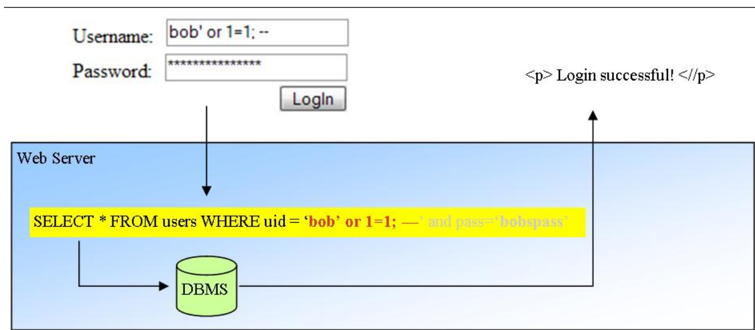
Results

- We designed a technique that allows us to detect ongoing injection attacks (and privacy violations). We call this technique GRASP.
- We analyzed the technique in depth and provided an implementation for PHP which is usable and secure: **GRASP for PHP**.

Simple SQL Injection Example



Simple SQL Injection Example



Demo

- We have two copies of the same PHP version. One of these is *GRASPED*.
- We will first install the original copy of PHP.
- Test a public exploit against a popular CMS (Bugtraq ID: 18492). The code of this exploit uses a timing attack to exploit a blind SQL-injection vulnerability.
- As a result, we will be able to retrieve the md5 hash for the admin user.
- Next we will install the *GRASPED* version of PHP. And test the exploit again. . .

Privacy Threats

- Web applications handle private data supplied by users.
- The lack of policies while disclosing this data (e.g., credit card numbers, medical history) can lead to privacy violation.
- Injection attacks can also lead to privacy violation through unauthorized data disclosure.

XSS - Cross site scripting

- They occur whenever an application takes data originated from a user and sends it to a web browser without validating or encoding it first.
- They are in fact a subset of HTML injection attacks.
- They allow the attackers to execute script code inside the victim's web browser.

Examples

- I subscribe in a discussion forum.
- I say my name is $< script > alert('Hi\ \ there!') < /script >$.
- While printing my posts...
- Everyone receives a pop-up with my message (imagine we could write any javascript program!)

Shell command injection

- They arise when the web application executes shell commands with user-supplied input (e.g., while working with directories and files).
- **User supplied data is passed, not well sanitized, to the shell interpreter**.

Examples

- Let's say I Upload a file with name "$any.txt\ \ ../\ |\ rm\ \ -rf\ \ ../|cd$"
- The script copies *any.txt* to parent directory, erases the entire parent directory and enters the (now inexistent) parent directory!

```
$filename = $_POST["file"];
$result = shell_exec("cp $filename ../uploadedFfiles");
```

Directory Traversal

- They allow an attacker to browse remote directories which shouldn't be allowed to.
- The vulnerability is often due to server misconfiguration (e.g., Apache's permissions on local directories).
- Sometimes the web application itself works with files directly and if a POST/GET parameter is tampered the attack can be successful.

Examples

- If the web server is not properly configured, simple URL rewriting can succeed.

  `http://www.myserver.com/images/thissite/../../../`

Command injection: Common principles

- The end user enters input not expected by the programmer.
- This input is not properly handled by the web application for its later use as part of a command / query / output.
- The attack becomes successful when the web application uses a second language for execution / output (e.g: SQL query, HTML output, etcetera.)
- The attacker is able to modify the web application's behavior through specially crafted input.

Known Attacks

- Multiple **PHPBB** injection vulnerabilities have been found in the last years.
- Popular CMS also had been compromised.
- Custom corporate applications are compromised every day.

Injection Attack Threats

- As we have seen, without the proper checks every web application can allow these attacks.
- The described attacks can result in:
  - Data loss/alteration/theft (*SQL - XSS - Shell injection*).
  - Content modification (*XSS*).
  - Defacement, e.g., site faking. (*XSS*)
  - Pivoting (to attack other web apps.)
  - Private data theft / usage.
- There is no silver bullet to prevent them...

Anatomy of an injection attack

- As described, the user enters specially crafted input through different attack vectors (e.g., form inputs, URL.)
- Web applications run inside an execution environment (VM / Interpreter).
- Three layers:
  1. Information is provided by the end-user.
  2. This information is improperly handled or sanitized inside the web application without being aware of the malicious data.
  3. Later, this information is used to perform operations that implement pre-designed functionalities, but specially crafted data turns those operations into attacks.
- Prevention/detection must be made in one (or all) of the former layers.

Anatomy of an injection attack

- Generally, detection of injection attacks is much more complicated than simple string search (e.g., checking for SQL keywords, '1=1'), for example, while requiring some encoding.
  - When tampering URL *GET* parameters.
  - When trying to bypass string escaping (e.g: PHP's addslashes).
  - When exploiting 3rd party APIs which may have binary bugs (e.g., extensions in PHP).
- Attack vectors change but the vulnerable targets (usually) remain the same:
  - Database engines (e.g., *SQL injection*).
  - Browser output alteration (e.g., *XSS*).
  - HTTP header injection.
  - We cannot elaborate a complete list!

Programmers' workarounds

- Escape, Encode, Filter Harmful characters inside user-controlled data.

Possible failures

- For example, while using regular expressions:
  - Case unsensitive RegEx sometimes can be bypassed using (upper—lower)case chars
  - Sometimes, depending on context, an attacker can inject %0d%0a (CRLF) followed by malicious data. Non-multiline RegEx only matches the first line, leaving the "tail" unchecked.
- Missing knowledge about string handling inside the programming language
  - ASP (3.0) for example, allowed %00 characters inside a string, a C-coded protection library may return a string is valid, but ASP continues using the malicious one.

Known Tools

1. Vulnerability detection.
2. Block & detect ongoing exploits.

Automated source code analysis

- White box testing (source code is required).
- Inaccurate. Less than perfect.
- Must be done before release. In development phase.
- False positive and false negative alarms.

Scanners

- Black box testing.
- Analyze a deployed application.
- Probe known vulns or have a fuzzer incorporated.
- High rate of false positives and false negatives.

Firewalls/IDS/IPS

- Work with known signatures or trained with stats, this induces false positives and negatives.
- Usually don't detect special crafted attacks (that involve only the targeted web app).
- They are susceptible to DoS attacks.

Summary

- Some of these tools require an exhaustive review of all the alarms.
- False positives and false negatives must be confirmed manually.
- They don't give evidence of the existence of a vulnerability.
- Once you run the aforementioned tools, you cannot be certain all the vulnerabilities have been eradicated.

Classical Taint Mode

Description

- It is a technique designed to address the problem of sending untrusted input to functions / operations that might be dangerous from the security standpoint.
- It is mostly used in development stages. Programmers can be aware of the checks they forgot to add, while an alarm raises every time tainted data reaches sensitive operations (e.g., shell commands, database queries).
- It can be used in deployment stage, after modifying the application according to detected alarms.
- According to the previously described anatomy, with taint mode, attacks are detected in the third layer.

Description

- It is bundled in some programming languages (either as extensions or directly bundled) such as Perl, Ruby.
- Strings are tainted as a unit. Either U: untainted, T: tainted.
- It gives the programmer the opportunity to check if he forgot a check before executing a sensitive operation (like shell command executions).
- If used for dynamic protection (while a web application is on line), produces a high rate of false positives while detecting on going attacks.

Examples

```
 my $cgi = CGI->new();
 my $user = $cgi->param('user');

 # $user is now tainted
 if ($user =~ /^(\w{1,6})$/){
      $user = $1;
      # $user is now untainted
}else
    ...
```

Pros & cons

```
SELECT * FROM users WHERE username='Bob'
```

- If the above query is fully tainted, it would raise an alarm although it is a valid SQL query.
- This behavior increases the rate of false positives.
- Once the alarm is raised, the developer must change his code accordingly.
- The string is treated as a whole, since no granular information is available.
- The system does not check whether a string is being sanitized or not.
- It only helps the developer in finding out where he forgot a sanitization.
- Useful in development stages. Can alert a developer of unhandled data.

Classical Taint mode
and
Character Granularity Information

Character Granularity Information

- String instrumentation adds a security mark to each character.
- This allows syntactic analysis over the string, augmenting precision.
- The previous query would be marked as:

  ```
  SELECT * FROM users WHERE username='Bob'
  HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHDDDH
  ```

  Where `D` means that character is marked as dangerous and `H` that it is harmless.
- The above query would not be considered dangerous.

Comparison: Classical Taint mode vs. Character granularity information

Classical

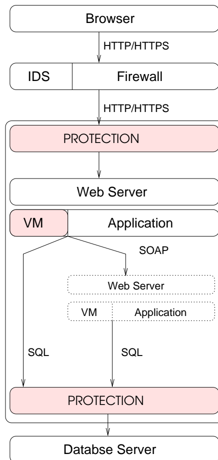
- Useful while in development.
- High rate of false positives while used in production / run-time.
- Application must be modified once an alarm has been detected.
- Already bundled with some languages.
- Sensitive functions & operations can be quickly detected.
- Low run-time / memory penalty.

Granular

- Useful while in development and while in production.
- Fewer false positives.
- High precision.
- Per-character information.
- Vulnerabilities can be precisely modeled allowing syntactical checks.
- Increased run-time / memory penalty (discussed later)

Introducing CORE GRASP

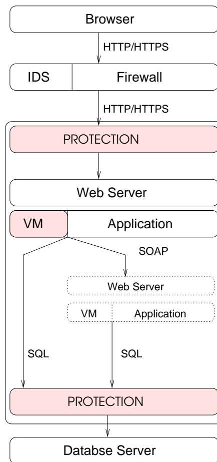
- Imagine a person reviewing each data entering / leaving the web server to / from users. Deciding whether it represents a threat for the web application or the user, marking entering data accordingly.
- This person can review each data inside / leaving the web server, sent to back-end APIs and servers, checking if they are indeed, a threat for the intended operation.



Preventing injection attacks

- GRASP replaces the reviewers role with three layers.
- First layer: between the web server and the outside world
  - Marks all incoming data from untrusted sources,
  - Prevents harmful data from being output to the outside world and
  - Enforces privacy policies.
- Second layer is inside the execution environment: it tracks the marked data inside the web application.
- Third layer is located between the web server and other servers (such as database servers):
  - Checks outgoing data.
  - Checks and blocks incoming attacks
  - Enforces privacy policies.

- Marks incoming data (e.g., database) as untrusted.



Design principles

- For each language we will modify the way any web application receives, handles and outputs data in order to prevent injection attacks.
- Security information is tracked inside the web application's execution flow.
- We will classify data operations / functions into three groups:
  - **Sensitive Sources**: Entry points to the execution environment where incoming data should be considered untrusted.
  - **Sensitive Sinks**: Functions / modules that forward / execute operations to / in the back end.
  - **Data manipulation operations**: Operations that handle or combine data.
- Each of these operations must be modified to be security-aware.

Architectural description

- For each programming language its execution environment is augmented to store additional security information about strings.
- Possible **Sensitive Sources** are:
  - *GET*, *POST* variables (e.g., while sending forms).
  - Environment variables.
  - Stored *COOKIES*.
  - *SESSION* variables.
  - Incoming database data.
- Possible **Sensitive Sinks** are:
  - Queries sent to database servers.
  - Data retrieved from a database that results in a later attack (database-stored XSS).
- All the attack vectors which can be controlled by a potential attacker are marked as dangerous while entering the execution environment through **Sensitive Sources**.
- Marks are propagated across **Data manipulation operations**.

Protection

- **Sensitive Sinks** protection is implemented as Finite State Machines.
- Granular information is used to detect and block attacks.
- They allow syntactic check for security.
    - There are less precise techniques such as keyword search.
- Well-formation of strings can be modeled deterministically for each family of vulnerabilities.
- For each API a checker (with different syntactic check) can be implemented.
- Inside the web server the protection is **the same** for all the web applications.

FSM - Example: SQL injection prevention

```
SELECT * FROM users where name = '' and 1=1;--'
HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHDDDDDDDDDDDDH
```

Security features

Injection attacks are prevented / detected:

- Existent web applications deployed on a server which incorporates the protection are protected immediately.
- A *FSM* checks for cross language boundaries threats.
- These *FSMs* are different for each family of threats (*SQL injection*, *XSS*, *Shell cmd injection*, *Directory traversal*, etcetera.)
- All detected attacks are logged.
- Neither source code modification nor ad-hoc checks are needed.

Privacy Features

- Privacy Policies are statements over the data which describe how a system should treat that data according to its source and destination.
- The solution provides the means for a “privacy officer” (PO) to define privacy policies stored in a configuration file.
- *Sensitive Sources* tag incoming data according to privacy policies.
- *Sensitive Sinks* perform privacy checks (which are different from security ones).
- *Data manipulation operations* propagate privacy tags.
- Data inside the execution environment is assigned a *privacy tag* e.g., public, store allowed, plain text, owner only, etcetera some of them can be defined by the PO.

Examples

- A database stores credit card information from users.
- This information should never be sent to the user's browser.
- A privacy policy could state: "Data from this column should never be printed" or "Data from this column can **only** be printed to its owner".
- While a web application sends content to a user's browser, this policy is enforced and no private data is sent unless it does not violate any pre-defined privacy policy.

Summary

- The former ideas can be directly implemented for PHP.
- We have developed a prototype that is a modification of the PHP interpreter.
- We chose PHP because :
  - Its source code is publicly available, so we could implemement it directly inside the execution environment.
  - Vulnerabilities in PHP-coded applications are disclosed **DAILY**.
- It is distributed as a patch to the PHP interpreter.
  - Installation has the same requirements as a typical PHP installation.
  - No special tunning is required.
  - No source code modification of existent web applications is needed.

Prototype: CORE GRASP for PHP

1. Covered functionalities.
2. Implementation details. PHP Internals.
3. Architecture description.
4. Modifications to the PHP interpreter.

Covered functionalities

- The prototype's last implementation was implemented for **PHP 5.2.3**. But we have been working on this technique since PHP 4.3.
- It implements marking for all **Sensitive Sources**.
- It only implements one **Sensitive Sink** protection: protects PHP applications against SQL injection attacks against MySQL databases.
- **String manipulation operations** were modified to propagate marks adequately (except RegEx propagation).
- Mark optimization was implemented.
  - For strings with all characters marked as *D* or *H* no character granularity is stored.
- Downloadable source code is published `http://grasp.coresecurity.com`. We expect to build a collaborative development community around this prototype.

Execution environment modifications

- As mentioned before, a modification to the execution environment must be made in order to allow security information to be handled.
- The basic data structure where data is stored inside the interpreter was extended for this purpose.
- These structures are the **zvals**. They store every value (& intermediate values) while inside the execution environment.

zvals

- The main component structure of zvals is the _zval_struct where we store our marks:

```
struct _zval_struct {
    /* Variable information */
    zvalue_value value;/* value union */
    zend_uint refcount;
    zend_uchar type;/* active type */
    zend_uchar is_ref;
    char *secmark;
};
```

zvals

- If the zval is a string we allocate the `secmark` to store per-character information:
  - `(char *)0` if the string has full safe mark.
  - `(char *)1` if the string has full unsafe mark.
  - `(char *)` pointing to an array of bytes, each one indicating a character's mark, while in mixed marks situation (safe and unsafe strings).
- *Optimization*: only in mixed mark situation double space is needed for the full string, otherwise 4 bytes are used.

Sensitive Sources

- We had to identify the **Sensitive Sources** and enable data marking while entering the execution environment.
- GET/POST/COOKIES are marked as dangerous while being loaded.

Sensitive Sinks

- Also **Sensitive Sinks** were identified, only one sink protection was implemented.
- While executing SQL queries (MySQL module) they are checked for attacks.

Data Manipulation Operations

- Per-character marks allow precise checking for vulnerabilities.
- Marks are propagated within string operations (concatenation, string functions.)
- Zvals are initialized with safe mark.
- Propagation allows tracking of user-controlled data.

Modifications

- Modifications to the interpreter were focused in:
  - Main data structures (*zvals, smart_str*).
  - Basic string operations.
  - Built-in string functions (*string.c*).
  - Zval constructor macros.
  - Zval initialization.
  - (*GPC*) variable initialization.
  - Execution environment files (*vm_execute, and others*)
  - MySQL module.

Testing

- We tested GRASP running popular CMSs, Database management interfaces and home-built web applications. It showed a ∼20% penalty in run-time.
- We tried to exploit known vulnerabilities with successful GRASP protection.
- We tested PHP's regression tests and added GRASP-specific ones.
- Memory usage is increased by ∼30% due to secmark allocation.

Other Protection Techniques and Improvements

- Other on-the-fly protection mechanisms (such as IDSs) also add run-time penalties.
- GRASP's overhead could be lowered.

Prototype specifics

1. Developer interface, added functions.
2. Configuration.
3. Logging capabilities.
4. Attack statistics.
5. Performance.

Developer interface

- Although protection is automatic several functions have been added as built-in to allow mark interaction:
  - *grasp_setmark()*: Sets full dangerous mark to the passed string.
  - *grasp_clearmark()*: Sets full safe mark to the passed string.
  - *grasp_getmark()*: Returns a string representation of passed string's mark.
- A developer could make use of these functionalities for example to:
  - Assure certain variables are consciously sanitized and should not be treated as dangerous.
  - Allow user-originated SQL commands, for example, while developing back-end applications.

Configuration

- Inside *php.ini* there are several parameters that can be configured:
  - *grasp_block_sql_attack*: Enables/Disables attack blocking.
  - *grasp_log_sql_attack*: Enables/Disables attack logging.
  - *grasp_log_sql_query*: Enables/Disables query logging.
  - *grasp_log_dir*: Specifies logging directory.

Two log files are available, *grasp_attack.log* and *grasp_queries.log*. For example, an attack would be logged as:

```
@Fri Jun 22 15:32:28 2007
192.168.21.12
/home/corelabs/src/testpatch/php-5.2.1-orig/tests/grasp/mysql_query.php
SELECT * FROM users WHERE username='' OR 1 = 1 OR '' = '';
....................................' OR 1 = 1 OR '' = '..
```

Where dots indicate safe marks and where the character is repeated it means it is controlled by the user.

Attack statistics

Core GRASP for PHP 5

http://www.coresecurity.com/corelabs

Grasp configuration	
grasp_block_sql_attack	On
grasp_log_sql_attack	On
grasp_log_sql_query	On
grasp_log_dir	/var/log/
Stats	
Total Queries	529
Total Attack	36
Attack percentage	6.00 %
Attack Details	
Total files attacked	6
[illegible]/php-5.2.1/tests/grasp/mysql_query.php:37	7
[illegible]/php-5.2.1/tests/grasp/mysql_query.php:87	3
[illegible]/php-5.2.1/tests/grasp/mysql_query.php:119	8
[illegible]/php-5.2.1/tests/grasp/mysql_unbuffered_query.php:30	7
[illegible]/php-5.2.1/tests/grasp/mysql_unbuffered_query.php:75	3
[illegible]/php-5.2.1/tests/grasp/mysql_unbuffered_query.php:101	8

- GRASP for PHP is being distributed as Open Source software.
- It is being released under Apache 2.0 license.
- We hope you can join us and collaborate with this project!

Future Work

Goals

We envision a lot of work for the future:

- Implementing protection against remaining attack vectors.
- Implementing full privacy protection.
- Polish current GRASP implementation.
- We envision optimizations suitable for high performance.
- Allow GRASP to work as a classic taint-mode to be used in development phase:
  - W. Venema proposed a taint mode to be used in development stage.
  - A common framework can be developed to allow both types of protection (classical and granular).
- Allow per-page protection policies.
  - This could allow an administrator to turn off protection depending on each page/site needs.

Ideas (cont.)

- Different levels of mark. For example: safe from XSS, unsafe for SQL injection, etcetera.
  - With 1 byte we could mask 8 different marks (1 bit per mark) even 2^8.
  - Privacy and security marks could be represented in one single byte.
- Implement GRASP for different programming languages (such as Java, ASP.NET).
  - Difficult but we started to think about the possibilities.
  - A possible way of doing this is by wrapping native string classes.
  - Modifying class loaders.

- **A Dynamic Technique for Enhancing the Security and Privacy of Web Applications**. Included in conference material
- **Enforcing Privacy in Web Applications**. Presented at *Privacy, Security and Trust 2005*. October 2005, New Brunswick, Canada.

Download available from http://grasp.coresecurity.com

Thanks!

ezequiel.gutesman@coresecurity.com
ariel.waissbein@coresecurity.com
ariel.futoransky@coresecurity.com