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

Ariel Futoransky, Ezequiel Gutesman and Ariel Waissbein[1]

Corelabs, Core Security Technologies. Humboldt 1967, Cdad. de Buenos Aires 1414, Argentina.

Abstract

Web application security and privacy became a central concern among the security community. The problems that are faced once an application is compromised necessarily demands special attention. The emerging programming languages, which allow unexperienced users to quickly develop applications, still fail to introduce mechanisms for preventing the aforementioned attacks. We introduce a technique for enhancing the security and privacy for a web-based solution, by augmenting its execution environment to include tracking information, that permits to efficiently identify and thwart several attack scenarios. The technique has been implemented to protect PHP, and could be extended to protect other web-development languages (such as Java, ASP.NET, Python, Perl and Ruby.) Typical exploitation methods as database-injection attacks, shell injection attacks, cross-site scripting attacks and directory-traversal attacks are prevented. Moreover, this technique prevents untrusted users from obtaining private data stored within the web application's network; thus, putting off the theft of sensitive data, as credit card information, as well as averting information leakage.

1 Introduction

During the last years, attack vectors have shifted from binary vulnerabilities to injection vulnerabilities in web applications. This is due to the fact that there has been a proliferation of vulnerable web applications and common components, and the fact that web programming languages fail to include the necessary prevention mechanisms.

Consequently, web developers need to carefully write their code embedding *ad hoc* prevention commands in order to counter even the most common (well-known) attacks ([SS02]). This turns to be a laborious practice that is prone to errors, and should be replaced by more robust development practices. Nowadays, new application security solutions have surfaced that promise to replace this work, handling enormous lists of attacks (e.g., [Ter05], [Imp05], [Air05]). Typically, these solutions imitate IDS techniques (i.e., act as proxies between users and web-based solution possibly combining signature-based recognition with statistical methods to stop attacks), achieving an insufficient level of security. Further, these solutions compile a suite of techniques aiming to avert a high-percentage of attacks. These techniques may reduce the risk, but will not avert some very dangerous attacks. Therefore, thorough security audits[1]are still required to discover and eradicate dangerous vulnerabilities from the code.

In the literature, dynamic solutions still require some additional considerations by the developer. For example, [BK04] addresses only database-injection attacks, it requires the developer to modify the code for the web-based solution and add unnecessary procedures. As a result this techniques produce a considerable slowdown, they are only probabilistically secure, and the modified scripts are more complex and then more difficult to audit. Parallel monitoring [LBW05] (see also [DG71]) are more general procedures

In this paper we present a security and privacy protection technique as applied to PHP. The method presented in this paper protects web-based solutions from certain well-known attack techniques that aim to steal or tamper with databases, and other core applications, run arbitrary commands in the web servers, damage other visitors, etcetera. It augments web development languages with additional features that (automatically) prevent these attacks. The method is deterministic, and one deploys this solution being certain of which attacks it diverts and can assert what attacks it prevents (i.e., what actions it averts) as it has a zero rate of false negatives and a very low incidence of false positives.

The scope of this solution is enormous, as a major portion of dot-com websites ($\sim$70% of the market) is developed in compliant languages (PHP, ASP, etcetera). According to statistical data borrowed from [Mit04]) (see also [Imp05]), our solution will prevent vulnerabilities present in over 80% of web applications found on the net. Further, it prevents the theft of credit card databases and averts other threats deriving from access to private information. We call this solution GRASP[2].

[1]Source-code audits is one stage of the development process, where the source code of the underlying program is checked for security mistakes.

[2]A copy of GRASP for PHP can be downloaded from http://www.coresecurity.com/corelabs.

2 Web-based solution threats

We start this section describing the typical web-based solution architecture, describing its players, and move on to give a high-level description of the attacks and privacy. The user experienced in web-based attacks and privacy threats can skip this section. For more information please refer to [JS06].

We will, hereafter, refer to web applications that are designed by web developers. Once deployed these web applications are accessed by users through their browsers.

A web application runs in several-tier infrastructures. When accessed by a user, a communication channel is created between the web application and the user's browser. A user sends navigation and data input (e.g., page requests, cookies) and other environmental information over this connection and is responded with web pages and cookies. Web pages are rendered by the user's browsers and cookies are stored in the users hard drive (depending on browser configuration).

The typical infrastructure for a web application is constituted by a front-end server, a middle-end server and a back end (a detailed description can be found at [Fow03].)

- The front end runs the graphical interface for users, letting the user access the web applications' functionalities. It will receive the user's input —through page requests— and pass them directly to the middle end. Once it receives answers (e.g., information) from the middle end, it generates web pages and sends them to users.

- The middle end (OSI layer 6) handles the logical operations of the web application. Once it receives commands passed by the front end, it will translate these requests to the respective API language and contact the API in the back end to retrieve this information. Finally, it will provide this information to the front end.

- The back end is constituted of computer systems that run computer programs providing the web application's core functionalities (the APIs). Say, for example, database engines, web mail servers, content-management systems, and business applications[3].

The front- and middle-end servers run the computer code developed in one of the so-called web-development languages. The code is thus called a

[3]Alternatively, less novel web applications consist only of a pair front end/back end. The front end for these web applications takes the role of both the front end and middle end as described above, while the back end keeps its role.

web-script or simply a script. Scripts are compiled/run on-demand by interpreters or virtual machines (VM for short). The main difference between scripting languages (e.g., PHP, Ruby, Python, Perl, ASP 3.0) and those who are executed by a virtual machine (e.g., Java, ASP.NET) is that scripting languages are interpreted (not compiled, as C-coded applications), and VM-based ones are compiled to an intermediate language. Another difference is that (usually) scripting languages allow a faster development cycle and for a novice developer it is easier to quickly develop a fully functional application in a scripting language rather than in a VM-based one.

2.1 Attack types

We consider input validation exploitation methods whereby user-supplied data is interpreted by user's browsers or one of the web-application's APIs violating the established policies, i.e., either where this data affects the syntactic structure intended for the front- and middle-end scripts that govern the APIs (examples follow), or where data that contains arbitrary cross language commands (e.g., SQL, HTML, Javascript, etcetera) gets forwarded to other users. These exploit methods, which include database-injection attacks, shell-code injection attacks, directory-traversal attacks, LDAP injection and cross-site scripting attacks, appeared years ago and continue to be present in most of today's web applications —as it is well documented (see, e.g., [vWW07], [CER00], [Rai98]). Evidence shows that the situation is worrying; often a "penetration test[4]" reveals that web-based solutions are vulnerable to exploits from this class and evidence [vWW07] shows an increase in the reported vulnerabilities.

It has been conjectured that the eradication, from today's web-applications arena, of these vulnerabilities that enable these attacks would automatically make most of these web-applications secure —according to statistical data ([Mit04]). Hence, the importance of this vulnerability class.

2.1.1 Database-injection attacks

Users communicate with the web application by sending URL requests and cookies. A database-injection attack is successful when one of these messages, containing arbitrary data, is parsed by front- and middle-end servers, transformed into a database command, and gets forwarded to the database engine where it gets executed. When the Database Management System's

[4]A penetration test, a computer security audit where the auditor attempts to obtain unauthorized access to the targeted network in order to assess its security.

(DBMS) query language is SQL, these attacks are called SQL-Injection attacks.

For example, let us pretend that the imaginary website `www.server.com` asks for its visitors to enter their name and password on the browser's input forms. Once this information is entered, the browser produces an URL of the form

```
http://www.server.com/login?uid=name&pass=****
```

which is sent to the web-application, where it is parsed, formated as a SQL command that checks if the user / password pair is valid, and then sent to the SQL engine. However, assume that Mallory enters

```
Mallory'; [malicious code]--
```

at the name input tag and an arbitrary password (where `[malicious code]` stands for an arbitrary database command). A vulnerability would arise if the middle-end server would parse the name as `Mallory';[malicious code]` and query the database with this name. Since the semicolon breaks the line in the SQL language, the SQL engine will execute the commands `select * from users where uid = '`*`Mallory'; [malicious code]--`*`;'`. (Italized characters are provided by Mallory.) The attack is then perpetrated.

A secure web application would analyze the input for an attack (e.g., and sanitize it) detecting, for example, the semi-colon or quote meta characters as "illegal" and preventing the middle-end server from sending these user-supplied meta characters to the SQL engine. However, different scenarios require different analyses. Also, these checks are difficult to program. Typically, a web developer will reuse the same checking algorithms for several situations and as a result the implemented checks might not be enough.

In order to exemplify this situation with a more complex example we can consider GET/POST methods, which are used to send information from the user's browser (e.g., while filling a form) into the web application. When this information arrives the VM/interpreter they are stored in two variables called GET and POST. Any end-user could modify these values (since they are originated in his browser), but if the web developer encodes for example, converting all characters to URL-encoding (hexadecimal) or uses a particular encoding to send GET/POST parameters (e.g., base64 with some secret key string) the attack vector must be revisited. Once inside the application, the encoded values would be eventually decoded in order to be used in a query. The attacker must then consider this kind of usage, encoding the malicious code in a certain way it bypasses developer's checks so the attack can result successfully.

2.1.2 Cross-site scripting attacks

"The essence of cross-site scripting is that an intruder causes a legitimate web server to send a webpage to a victim's browser that contains malicious script or HTML of the intruder's choosing ([Raf01])" Cross-site scripting attacks, XSS for short, can be traced to a CERT advisory in early 2000 ([Cen00]). A pragmatic example is that of guest books, visitors (users) will submit input data to the web application which is later included in the web application's repository and presented to other users. If this data is not properly sanitized before being forwarded to end users, an attacker may be able to post in the forum arbitrary text containing HTML tags or script, which would result in the unintended execution of scripts on other visitors' browsers[5].

These attacks are one of the most dangerous, yet common vulnerabilities, which have a devastating impact over the affected target. A successful attack may result in the execution of arbitrary code (script code) inside another users' browser, leading to credential theft or impersonation. For example, an attacker could exploit a XXS attack in an etailer web application so that any user that connects to the etailer will execute a script while buying certain products using their own credentials (username / password / credit card), but sending the products to the attacker's home.

2.1.3 Shell-injection attacks

In some web applications the middle end server will access a computer in the back end and run shell commands (e.g., in order to access mail services or handle files.) These commands may take parameters provided by the web application's users. These parameters need to be checked and sanitized. Shell-code injection attacks occur when the attacker provides parameters for a shell command, that contain malicious code, and the parameters are forwarded (and executed) without being checked. For example, if an attacker executes this attack he can compromise the back end and thus all the applications running in it.

2.1.4 Directory-traversal attacks

A web application may require to access the file system in the back end, e.g., while some user is submitting a file. These accesses, are made by provid-

[5]Of course, visitors might disable the scripting functionality in their browsers. However, this poses a usability burden to users that typically choose to enable scripting.

ing certain parametric information in URL requests or cookies. Directory-traversal attacks occur when an attacker is able to access restricted directories or execute arbitrary commands in the file system. As a result, he can compromise the entire back end server.

2.2 Privacy threats

Web-scripting languages, as those cited above, do not include any mechanisms to distinguish sensitive information of what an attacker may take advantage of a security glitch in the application business-logic or presentation layer and steal, delete or manipulate sensitive information.

Web applications typically access sensitive data through some applications in the back end for processing, however this data should never be available to users. For example, credit card information for the clients of an etailer will be available to the business application in charge of processing sale transactions; it might also happen that the personal credit card information is partially available to its owner during his visit to the web application; however, no user should be able to access credit card information for other users.

On the other hand, it sometimes happens that users are able to access debug data, which should only be available to developers. Attackers can then use debug data to learn structural information of the web application which might help them to craft an attack. Hence, the threat of not restricting the access to debug data.

2.3 Countermeasures

While trying to protect web applications, during development, from the afore-mentioned attacks, developers include checks and workarounds specially designed for the affected web application. These countermeasures include: harmful character escaping (such as $', '', \backslash, ., *,$), URL encoding, harmful character filtering, regular expression (hereafter RegEx) matching for allowed input, and others.

These workarounds present different weaknesses. RegEx validation can lead to a false sense of security since if a developer fails to consider all the possible syntactic formations for a given input, the security check tends to fail. Most of the times RegEx validation requires careful attention and forces the developer to take into account lots of variants. Sometimes (due to developing deadlines) checks are not properly designed. As an example, let us consider case unsensitive RegEx which can be bypassed by using lower/upper case input. Sometimes depending on the programming language, %0d%0a (carriage return, line feed) can be inserted in the middle of a string, thus, non-multiline RegEx would validate the string's prefix, ignoring its tail. Another example we can consider is the known case of strings in ASP 3.0 applications, which allow %00 characters among valid characters. A c-coded protection library would check the string until it reaches the malicious character, reporting it is safe as it reached end of string, but the ASP application will still be using the malicious string. As every workaround is designed only for the affected application it does not provide a definite solution to the injection vulnerabilities problem.

Other countermeasures that can be considered as testing tools include source-code auditing, running vulnerability scanners, and testing the developed applications with IDSs/IPSs, but they also fail in completely solving the problem.

3 Grasp for PHP

We implemented a modification of the PHP interpreter which incorporates protection for injection vulnerabilities and a mechanism for enforcing privacy policies. It replaces the original PHP interpreter and the installation procedure is the same as the one required for a normal PHP installation (see [PHP]). Once installed, it starts protecting all deployed web applications without any source code modification.

The implemented security features allow the modified interpreter to detect and block attacks on the fly. This is achieved by deterministic security checks based on data security information. Privacy policies can be defined by a security officer responsible for the site (e.g., declaring some stored data is private, public, accessible by owner only, etcetera) being automatically enforced at run-time.

We first explain how does the technique underlying our solution work and show that it prevents the previously described attacks. We then give details as to implement it, estimate its performance, and end this section's discussion on the security and performance given by a prototype we implemented for PHP that runs in all platforms (i.e., win32, all BSD, and most Linux distributions) and handles MySQL databases.

Briefly speaking, Grasp stores the source of each data entering the interpreter/VM (e.g., users, databases) and uses this information to perform checks before using it in a potentially dangerous context (e.g., sending queries to a database engine.)

The data operators/functions inside the interpreter/VM can be divided in three groups: sensitive sources, sensitive sinks and data manipulation operations. The sensitive sources are those functions or entry points to the execution environment (VM/Interpreter) where data should be considered untrusted, for example, when retrieving data from GET/POST/COOKIES, all data can be controlled by a potential attacker, though, our prototype treats as dangerous all data incoming from these sources. Data manipulation operations are those operations inside the application which handles and combines data. Finally sensitive sinks are those functions/modules that forward and execute operations in the back-end (e.g., database queries, API communication.)

Every character received and processed by the modified script interpreter is labeled, character by character, with any combination of the labels *untrusted* or *private/trusted*. Both user-supplied data and user-controlled data within the back end (e.g., those fields within database that store user-supplied information) are labeled as untrusted; the script interpreters will also access an initialization file that stores which fields in each database contain private information and mark this data as private for the VM. On the other hand, the modifications in the interpreter/VM will propagate marks after any manipulation on local variables, i.e., the result of every data manipulation operation, that is affected by local data containing untrusted characters, is labeled as untrusted and the result of every operation that is affected by local data containing characters labeled as private/trusted is labeled as private/trusted (more details follow in the next section) keeping always per-character information.

For example, if we represent variables inside the interpreter as a tuple $(\texttt{type}, value, < length >, < securityMark >)$, we can consider the concatenation function that, given two local variables computes their concatenation, which is modified to propagate marks, as follows: Say, for example that a=(`char`,“hello”,5,.....) and
b=(`char`,“;kill app”,9,XXXXXXXXX), then their concatenation is
a||b=(`char`,“hello;kill app”,14,.....XXXXXXXXX). With `X` meaning that character's mark is untrusted and . trusted.

To avert *database-injection attacks*, a module will prevent the middle-end server from sending the database engine commands containing characters marked as untrusted that could affect its syntactic structure. More explicitly, once the command is parsed by the middle end server, but before it is sent to a sensitive sink (e.g., database engine), it is parsed and analyzed by the modified VM so that all meta characters[6]that are labeled as untrusted are blocked. To avert *shell-injection attacks*, a module will block the middle end from sending special meta characters as commands to the shell API sensitive sink, i.e., commands comprising untrusted meta characters are blocked and only inoffensive parameters are permitted. To avert *directory-traversal attacks* a module will block the middle-end server from sending untrusted characters, comprising special meta characters, to the file-system API sensitive sink. To avert *cross-site scripting attacks* in our method, a module will prevent the front end to send scripting tags and code, to users, if they contain characters labeled as untrusted.

The quality of these checks can be measured by the false positive and false negative alarms they provide or fail to provide, and it depends on how accurately does the analysis check for an exploit. In the above cases, it appears that the extra information provided by the marks (which should be 100% accurate) is enough to describe known attacks. In our prototype we modified the MySQL module to perform specially designed checks. These checks take into account per-character security labels and the specific SQL language structure so it can distinguish a dangerous query before it is executed.

One may also log occurrences of any of these potentially-dangerous actions in order to enrich forensic information. Furthermore, our method does not limit to forestalling the four above-mentioned vulnerabilities, but may prevent others which fall in the category we defined in Section 2.1. Namely, our method allows the developer to define, for each new API, the tools for parsing and analyzing the input for this API (e.g., this analysis profiting from the augmented input information), and therefore decide whether to let pass or block these inputs, or whether to log them. On the other hand, this logging capabilities can be used to obtain signatures[7]of new attacks.

The protection mechanism for injection attacks can be modeled by a Finite State Machine (FSM for short) which allow a formal representation of well-formed strings. The FSM evaluates a predicate and then answers true if the string does not represent an exploit, and false if it does. We can design a FSM for each kind of vulnerability, allowing a precise per-character analysis

[6]Meta characters are those characters which have a "special meaning" in the underlying language API. For example, ";" and quotes are SQL meta characters since in the SQL language semi-colon means that the command line finishes and quotes are for enclosing string constants. With our method, meta characters are defined once and for all per language in a special configuration file.

[7]A signature is a string which identifies the structure of a malicious input. For example a string beginning with a single quote and ending with a double dash can be considered an attack (of course this is not precise.)

in order to perform security checks detecting vulnerabilities in cross language boundaries (e.g., SQL inside PHP, Javascript inside HTML, etcetera.)

Another aspect of our method will provide the privacy-enhancement capabilities [FW05]. On setup, the system owner (or the developer) can label sensitive fields within each database as private, it will also define the private directories in the file system, defining if a given action can be performed over that data (e.g., the field credit card can never be sent to the users' browser.) This information is stored in an write-protected configuration file. Once the web application is deployed, a module will mark as private all characters that enter the VM and come from these private "data sources", and another module will block characters marked as private from being sent from the front end to the user's browser (e.g., forwarding credit card numbers to the user's browser). These actions, again, can be logged.

3.1 Prototype

Our current prototype is an instrumentation of PHP 5.2.1 which implements SQL injection protection against MySQL databases. The FSM for this protection was based on MySQL's lexical analyzer. This allowed us to develop a security-aware query checker.

In order to cover sensitive sources, sensitive sinks and data manipulation operations we modified PHP's interpreter. We will now describe some of the modifications that we made.

One of the key decisions on the prototype was where to store per character security information. One of PHP's central data structures are zvals. Every variable inside the interpreter is represented by a zval. We decided to augment this structure by adding two values, which we call secmark and secinit. The former stores the mark for the string and the latter is for integrity purposes (it acts as a magic number that determines zval's security mark). A naive implementation would cause a doubling in the memory required so the mark storage was optimized to prevent this while storing a string.

Sensitive sources were covered by marking as *untrusted* all data incoming from GET/POST/COOKIES (GPC for short) and every time a script loads GPC variables, they automatically become marked as *untrusted*.

The only sensitive sink implemented was in the MySQL module. We developed a FSM inspired, as we said, in MySQLs lexical analyzer. We then instrumented the MySQL module by prepending the FSM check before each command is sent to the MySQL engine.

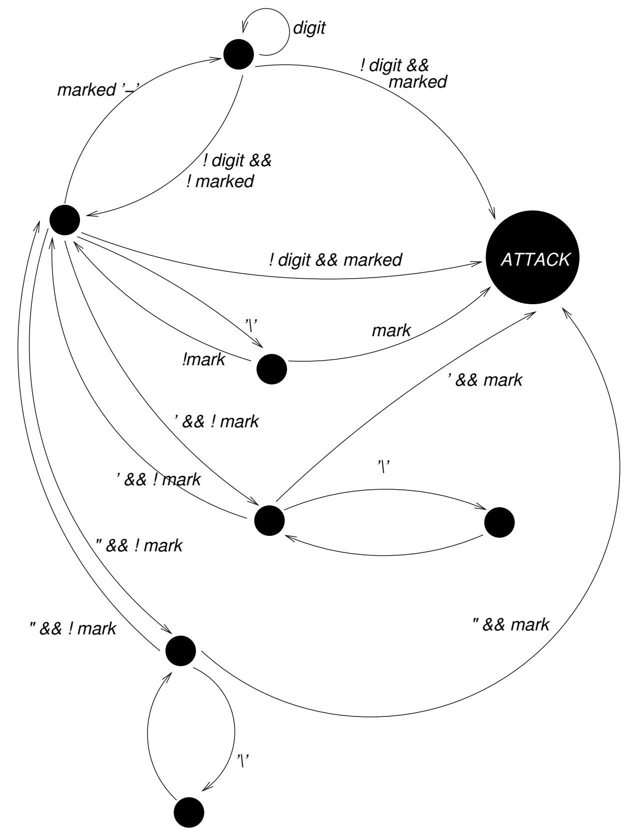

Figure 1: SQL check FSM. The leftmost state is the initial state. Each transition occurs when a new character is analyzed. The only state which causes the system to block the query is the ATTACK state. Any other state is considered safe.

Data manipulation operations are of course the central part of the implementation, since it has to ensure accurate mark propagation inside the interpreter. The main modifications were made in functions that create/destroy/ copy zvals and inside the native string operations (such as implode, explode and concatenation, not all of them in this prototype). Another important modification of PHP's implementation is the concatenation functions, implemented in a module called smart_str, which handles all types of concatenation (string to string, char to string, etcetera).

The prototype also includes new functions to the PHP core. Although the prototype's protection is automatic, it provides several functions that could allow a developer to interact directly with security marks: *grasp_setmark* sets full untrusted mark to the passed parameter, *grasp_clearmark* sets full trusted mark to the passed parameter, *grasp_getmark* returns the parameter's security mark represented by a string. Finally, a statistic function was added *graspinfo* which returns a summary including configuration parameters, attack rates and a detailed list of the attacked files lines inside them where the attack took place.

Several configuration parameters can be adjusted. Logging and blocking capabilities can be disabled independently and allow different combinations of logging attacks, logging queries and blocking attacks.

3.2 Implementation and performance

Our method was implemented in PHP5 version 5.2.1 with protection for MySQL databases. We also developed the protection for earlier versions 4.4.3, 5.0.4, 5.0.5, 5.1.2 both for Win32 and Unix/Linux systems.

Since PHP is open source software, its source code is available and one can methodically go through the labyrinths of lines in order to modify every macro in the virtual machine. Cautious auditing and testing will confirm this.

GRASP for PHP requires only a brief install, almost identical to the classical PHP install. The private information functionalities will start working as soon as the configuration file contains the private specifications.

We tested GRASP for PHP by launching the 10 latest SQL-injection exploitable vulnerabilities against a vulnerable (unpatched) version of PHP, protected by Grasp: as a result we confirmed that no exploit was successfully deployed.

Web applications deployed on the Internet running on Grasp for PHP performed smoothly, only a minor efficiency loss was verified in normal program behavior. Additionally, the logging capabilities were helpful to identify development errors in one web application, which could be exploited before GRASP was installed.

Stress tests were performed and showed (as expected) a double overhead while performing a very big amount ($\approx$10000) of queries in a row with mixed database query strings, i.e. strings composed by both dangerous and safe characters (which demand double work while copying strings). This is not a common situation, and full-marked (dangerous and safe) database query strings showed a 30% penalty in run time.

4 Other Implementations - Future Work

Implementing this protection technique over other web development languages might be possible but requires a careful design. For example, for ASP we could encapsulate the underlying virtual machine with a GRASP module (cf. [NS05]).

Although we did not implement them, on languages like ASP.NET or Java, the technique can also be implemented through reflection, and by wrapping the class loaders of native classes and data types, replacing them with mark-aware versions.

Other uses can be devised for this technique. A classical taint-mode can be used by developers while in development phase. [Ven06] proposed it should be useful such a mode and we are envisioning future collaboration.

Grasp's attack blocking can be disabled, turning it into a very powerful logging tool. This logs can be used to feed log analyzers. This would help in forensic activities and also might help in testing during the development cycle.

Per-page protection is another feature we think would be very useful. Grasp could allow a server administrator to define per-page protection rules. For example, disable blocking for certain back-end pages which need to execute user-provided queries.

In privacy means, we could manually define privacy policies over fields directly in the database. Allowing the system to store privacy policies available for different web applications in different web servers that connect to the same database.

We could also want to automatically determine data's security (e.g., by analyzing valid queries sent to a database with data marked as dangerous) and set which database columns must be considered as dangerous, automatically generating a the security marks for this column while being selected into the web application.

As we publish this article, Grasp's source code is being released for free use open source, hoping interested people in the community can join us in completing and augmenting the solution, since it has demonstrated its effectiveness and ease-of-use.

References

[Air05] Airlock. Web-application security, January 2005. URL: `http://www.seclutions.com/en/ct_products_en.htm`.

[BK04] Stephen Boyd and Angelos Keromytis. SQLrand: Preventing SQL Injection Attacks. In Markus Jakobsson, Moti Yung, and Jianying Zhou, editors, *Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, volume 3089 of *LNCS*, Yellow Mountain, China, June 2004. Springer.

[Cen00] CERT(R) Coordination Center. Cert advisory ca-2000-02 malicious html tags embedded in client web requests., February 2000.

[CER00] CERT. CERT Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests. `http://www.cert.org/advisories/CA-2000-02.html`, 2000.

[DG71] P. Deutsch and C.A. Grant. A flexible measurement tool for software systems 71. In *Proceedings of the IFIP Congress*, pages TA–3–7–TA–3–12, Ljubljana, Yugoslavia, 1971.

[Fow03] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.

[FW05] Ariel Futoransky and Ariel Waissbein. Enforcing privacy in web applications. In *Third Annual Conference on Privacy, Security and Trust, October 12-14, 2005, The Fairmont Algonquin, St. Andrews, New Brunswick, Canada, Proceedings*, 2005. URL: `http://www.coresecurity.com/index.php5?module=ContentMod&action=item&id=1385`.

[Imp05] Imperva Software. Total application security, January 2005. URL: `http://www.imperva.com`.

[JS06] Caleb Sima Joel Scambray, Mike Shema. *Hacking Exposed Web Applications, 2nd Edition.* McGraw-Hill Osborne Media, 2006.

[LBW05] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, Feb 2005. To appear.

[Mit04] Robert L. Mitchell. Q&A: WebCohort's Shlomo Kramer on the app-layer battleground. *Computerworld*, February 2004.

[NS05] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *12th Annual Network and Distributed System Security Symposium (NDSS '05), Proceedings*, San Diego, California, USA, February 2005.

[PHP] PHP Installation requirements. Php's official site. URL: `http://www.php.net/manual/en/install.unix.php`.

[Raf01] Jason Rafail. Cross-site scripting vulnerabilities. Cert Coordination Center, Technical Report, 2001.

[Rai98] Rain Forest Puppy. NT web technology vulnerabilities. *Phrack Magazine*, 8(54), 1998.

[SS02] Joel Scambray and Mike Shema. *Web Applications (Hacking Exposed)*. McGraw-Hill Osborne Media, 2002.

[Ter05] Teros. Web-application security and performance, January 2005. URL: `http://www.teros.com`.

[Ven06] Wietse Venema. Php internals mailing list, Dec 2006. URL: `http://www.mail-archive.com/internals@lists.php.net/msg25405.html`.

[vWW07] Andrew van der Stock, Jeff Williams, and Dave Wichers. The ten most critical web-application security vulnerabities (2007 update). OWASP technical report. URL: `http://www.owasp.org/index.php/Top_10`, 2007.