Detection and Prevention of Insider Threats in Database Driven Web Services

Tzvi Chumash and Danfeng Yao*

Rutgers University, Computer Science Department,
110 Frelinghuysen Road, Piscataway, NJ 08854, USA
{tzvika,danfeng}@cs.rutgers.edu

Abstract. In this paper, we take the first step to address the gap between the security needs in outsourced hosting services and the protection provided in the current practice. We consider both insider and outsider attacks in the third-party web hosting scenarios. We present *SafeWS*, a modular solution that is inserted between server side scripts and databases in order to prevent and detect website hijacking and unauthorized access to stored data. To achieve the required security, *SafeWS* utilizes a combination of lightweight cryptographic integrity and encryption tools, software engineering techniques, and security data management principles. We also describe our implementation of *SafeWS* and its evaluation. The performance analysis of our prototype shows the overhead introduced by security verification is small. *SafeWS* will allow business owners to significantly reduce the security risks and vulnerabilities of outsourcing their sensitive customer data to third-party providers.

1 Introduction

As e-commerce becomes more common on the Internet, an increasing number of small businesses (e.g., online stores) use hosting providers to open their doors to online customers. These small businesses put their trust in various hosting service providers for the benefits of higher availability, fast website access, round-the-clock support and a very low cost [6]. As a result, customer data, which may contain sensitive information (such as Social Security numbers or credit card information), is either stored by these third-party providers, or can be accessed from their servers.

In what follows, we will use the following terminology to distinguish the entities that concern us. *Service provider* refers to an organization and its employees which are in the business of leasing web service resources. *Web-server* refers to a machine and software running on it that is owned by a service provider[1]. A web-server provides website content to an end-user. *Website content* refers to HTML pages, scripts and any data in a database that is stored or served by the scripts on a given website. An *end-user* or a *customer* is a person that is interested in website content for a given website,

* This work has been supported in part by NSF grant CNS-0831186 and the Rutgers University Computing Coordination Counsil Pervasive Computing Initiative Grant.

[1] Software refers to the entire non-hardware environment provided, including the operating system, web-server software (such as Apache) and PHP interpreter.

this person may disclose sensitive information to this website. A *website owner* is the person (or organization) that is in charge of website content and has interest in keeping end-user information safe. Web site owners lease web-servers from service providers in order to run their web site.

A website owner may purchase a (low cost) certificate to authenticate their establishment [5, 7]. When customers fill out HTML forms with their sensitive information on SSL protected websites, they are led to believe that with the padded lock icon and an authenticated signed certificate, the data that they are about to hand over is safe. However, SSL only protects the *end-to-end* security of the data from the customer's computer to the web-server [5], it bears no indication of the kind of protection that the data gets once it enters the domain of the service provider. As we will explain next, many website owners typically store the database credentials in clear-text. This information can be easily used to login to the website owners' databases and either retrieve or alter sensitive information of customers.

Many websites are driven by database systems, as databases are widely used to store customer data and product information and play a crucial and central role in modern e-commerce. Using a combination of server-side scripts[2] and database connections is a widely used approach in providing dynamic content to online users and in retrieving and storing customer data on databases. These scripts offer website owners a versatile interpreted language that can create complex web environments, with libraries offering connectivity to many other services such as databases. However, there has not been any framework provided within the web-server environment to allow for safe execution of server-side scripts. Server-side scripts are called by web server modules, which are initiated by end-user requests. Due to the interpretive nature of server-side scripts, and with the tools available today, it is impossible to obscure or hide sensitive program details from users with administrative access to the web-server machine. Furthermore, aside from being completely readable, these server-side scripts can be altered by privileged users without the owner's consent or knowledge.

Storing database passwords as *clear-text* in server-side scripts is the *de facto* practice for IT professionals world-wide. In an outsourced setting, this approach implies that it is extremely easy for malicious employees at the service provider organization to access the passwords of business owners and thus their customer data. Security breaches at providers, caused by outside adversaries, may also expose hosted sensitive information. However, neither the research nor the industrial community have been giving enough attention to protecting sensitive data from being used by unauthorized parties and untrusted service providers in the outsourced setting.

Server-side script creators have been aware of the clear-text credential issues [4, 19], but have been concentrating on the ability of an end-user to see them, in case the web-server software is badly configured or compromised. The service providers and their web servers have been assumed trustworthy, therefore no protection against insider threats with respect to server-side scripts has been provided. The existing solution to protect against credential disclosure to end-users is to create an external script containing the clear-text credentials, placing it outside of the document root, and including it by the called script. While this naive hiding strategy might protect against poor

[2] Such as PHP, ASP, PERL, Python, etc.

web-server software configuration, a tampered version of a web server, as well as people with access to the server's operating system, can read that file. Another approach is to include the credentials inside the web server configuration file [19], but this again does not protect against insider threats.

Our Contributions. In this paper, we take the first step to address the gap between the security needs in outsourced hosting services and the protection provided in the current practice. We consider both insider and outsider attacks in the third-party web hosting scenarios. We describe the threats and security vulnerabilities that exist in today's web environments. And finally we present *SafeWS*[3], a system with a relatively low overhead to mitigate the threats of potential security breaches at service providers. *SafeWS* is a novel and modular solution that prevents and detects website hijacking and unauthorized access to stored data.

To achieve the required security, our solution utilizes a combination of lightweight cryptographic integrity and encryption tools, software engineering techniques, and security data management principles. Our solution is written in C/C++ and contains a component that resides between server-side scripts and database systems, as well as components that reside off the service-provider's server. We evaluated *SafeWS* with PHP scripts and the performance analysis shows the overhead introduced by *SafeWS* is small.

SafeWS allows business owners to significantly reduce the security risks and vulnerabilities of outsourcing their sensitive customer data to third-party providers. By deploying our solution, website owners can provide their customers with a robust and secure storage of their sensitive personal information.

The main goal of this work is to improve the protection of outsourced sensitive data on untrusted web servers. We believe that low-budgeted database driven websites that use shared-hosting have the greatest risk for unauthorized disclosure of information, and therefore designed *SafeWS* for their needs. Our framework enables website owners (e.g., small business owners) to automatically verify the integrity of service providers and their web servers. Most importantly, we efficiently and effectively prevent sensitive credentials, e.g., database locations, names, usernames and passwords, from being stored as clear-text at the service provider side. The highlights of our solution are shown below.

- Guarantees that only authorized webpages from the web server can access a database that stores end users or product information.
- Effectively hides database access credentials from web server administrators who can read any file on the system.
- Ensures the integrity of outsourced websites by detecting and monitoring suspicious environments and activities; provides website owner with notifications of suspicious activities.

SafeWS demonstrates a general security design principle for outsourced computation that is a contribution beyond the specific server-side script problem studied. Our work

[3] SafeWS stands for Safe Web Script.

can improve the security of any kind of database driven web server system in a third-party service provider setting.

Scope of our work. We decided to concentrate our work in the layer between server-side scripts and databases as that is the intersection where the end-user, the owner and the service provider meet. It allows us to verify the authenticity of owner-built scripts and the healthiness of the service provider environment, while also protecting end-user data from being maliciously used. At this layer we can also notify the owner of any foul play, thus minimizing the possible impact of an attack. Authorization and access control, if implemented within server side scripts, can also benefit from *SafeWS* as long as a database connection is used, though *SafeWS* is not meant to be an access control system. We also concentrated our efforts on preventing data theft and corruption, rather than handling denial of service attacks. Lastly, we decided to evaluate our system on a platform (LAMP[4]), that is widely used in the third-party hosting setting, is open-source and easily implemented, but due to the generality of our design, we believe our system to be valid for any server-side scripting language (*SafeWS* can also improve the security of non-scripting programs such as compiled/binary server-side programs).

Organization of the paper. The rest of the paper is organized as follows. Our adversarial model, security definitions, and assumptions are described in Section 2. The architecture and protocols are presented in Section 3. Our security analysis is discussed in Section 4. Performance evaluation of *SafeWS* is described in Section 5. Related work is given in Section 6. Finally, we describe future work and conclude in Section 7.

2 Definitions and Trust Models

In this section, we give the necessary definitions, trust model, adversary model, and security definitions used in our solution. First, let us briefly recapture our definitions on the types of entities introduced in Section 1.

- *Service provider* refers to an organization and its employees which are in the business of leasing web service resources.
- *Web-server* refers to a machine and software running on it that is owned by a service provider.
- An *end-user* or a *customer* is a person that is interested in website content for a given website.
- A *website owner* is the person (or organization) that is in charge of website content and has interest in keeping end-user information safe.

In reality, website owners typically want to provide a cost-effective solution to their customers. Most of them are unaware of or unable to comprehend the security requirements and guarantees in the outsourcing environments of service providers. A large number of website owners do not write server side scripts for data manipulation themselves. Instead, those functions are provided as part of the outsourcing service.

[4] Linux, Apache, MySQL and PHP.

Similarly, website customers are usually completely unaware of the business outsourcing agreements between website owners and service providers. Consequently, the end-users assume that their sensitive information is only released to the website owner, and no one else[5].

Trust Relationships. Our trust model is simple and intuitive. The main interactions are between the website owner and the service provider. Website owners are not malicious. Web servers are not trusted by the website owners, and therefore, our solution is used by a website owner to verify the integrity of the outsourced environment. Website users trust that the website owner is ensuring the outsourced web server is not compromised.

Adversarial Model in SafeWS. Instead of assuming abstract adversaries, we strive to give a concrete and comprehensive categorization of types of attackers, from both inside or outside the service provider organization. Such a practical analysis is both crucial and fundamental to the security of proposed solutions. Because of the specific application scenario studied, we are able to describe a concrete adversarial model.

We divided the security threats on a hosted web server into different levels based on the position and experience of the possible attacker. While we would want to believe most people would not violate the trust put in them by their employer, all it takes is one person with a different set of motives. Every given level is assumed to encompass the abilities of the previous level. Thus, a novice hacker may do anything that an administrator could do.

We believe that some of these security threats can be reduced by making them more difficult to accomplish, as well as providing our own threat of recognizing an attack when it happens, and notifying the owners.

- *Nosy Administrator.* Any server administrator with super-user access can scan the directory tree for clear-text server-side script files. Upon finding those files, this person can then read the database credentials, and learn the names of the columns and tables where sensitive information is stored. This person can then decide to enter the provided database and look at the information stored there. While this person might be acting out of curiosity, the outcome is that an unauthorized person was able to view secret data.
- *Disgruntled Employee.* A disgruntled employee, or specifically a disgruntled employee with system access, may maliciously obtain information from hosted client websites in the same way the Nosy Administrator would, but for different reasons. This person can actually cause financial harm by disclosing information to outsiders, or using it for personal gain.
- *Novice Hacker.* In addition to all the threats defined above, a novice hacker may attempt to hijack a website by changing the server-side script files that obtain web-user information, or by adding new script files. A novice hacker may also try to replace the web-server executable with a malicious one.
- *Advanced Hacker.* An advanced hacker may analyze traffic in and out of the system, change the kernel, scan the memory, and reverse engineer any program.

[5] For server authentication, users can rely on their web browser indications, such as the lock icon, an https address and a valid certificate.

Definitions of Security. We formalize our security goals in three requirements, namely, *secrecy of database credentials*, *provenance authentication for database access*, and *integrity of outsourced environments*.

The *secrecy of database credentials* is defined as that the credentials (e.g., passwords) required to access the website owner's database need to be confidential and cannot be learned by privileged users on the service provider's machine, who can read *any file* on the system. We give the web server administrators significant amount of power, which is necessary in this outsourced scenario. This requirement implies that the website owner's database needs to be maintained by a different provider from the web server provider. In practice, such a separation of duties principle (i.e., separating web server provider from database provider) is in general desirable to constrain and balance providers' privileges.

Provenance authorization for database access is defined as that only authorized webpages on the web server can access a database that stores end user or product information. The *provenance of a database request* is the webpage that initiates the database connection. This requirement means that the provenance information associated with a database request must be recorded, submitted, and verified before a request can be satisfied.

Integrity of outsourced environments is defined as that any tampering with the web and computing environments, including parameters, software, and libraries, should be detected and the website owners' notified.

Security Assumptions. The owner of a website can obtain the SHA-1 hashes of non-hacked Apache executable and modules running on the web-server before the server is compromised. The owner of a website has a separate, non-compromised machine where he can store private keys, authenticate server-side script files, periodically activate configuration scripts and compile *SafeWS*. The machine running *SafeWS* may be compromised after *SafeWS* is already in place. The service provider may, from time to time, upgrade the software on the web server. It is up to the owner to keep up with these changes and reconfigure *SafeWS* accordingly (as automatically identifying whether a change of a library or executable is done maliciously or not is outside our scope). If multiple script files which require database access include each other, it is up to the owner to ensure each of them calls *SafeWS* as we do not want to introduce the overhead of nested source files and pre-compilation to our run-time environment. Embedding connection strings in compiled code is an approach that can provide added security, as opposed to just placing them in clear-text scripts, as some work needs to be done to decompile and evaluate the data. This is common practice for any compiled (non-script) DB accessing CGI[6] on a web server, but it is not flexible, nor secure enough. Recompilation is needed whenever credentials change, as the environment is not authenticated and the credentials are revealed after one decompilation.

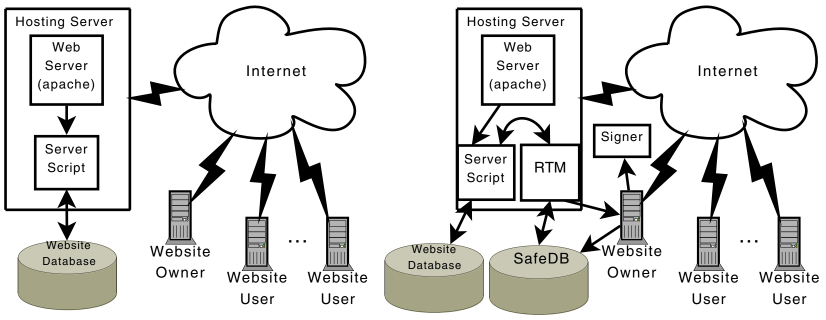
3 Architecture

Intuitively, our solution provides the website owner a way to evaluate, assess, and authenticate the working environments of a web server hosted by a third-party service

[6] Common Gateway Interface.

provider. With our solution in place, a website owner (or his trusted technologically savvy agent) stores and encrypts part of the security information used for authentication on the third-party web server. If abnormal conditions are detected, our solution has the ability to automatically notify and alert the website owner and users to the well-being of the third-party server. Next, we will give detailed descriptions on the architecture, components, and procedures of our solution.

The design of our solution is divided into two parts. The first part includes all actions needed during run-time to ensure only authorized and authenticated scripts may access the database, and other attempts would cause notifications to be sent to the owner of the site. The second part is an offline process that happens mostly outside the server to ensure proper configuration of the solution[7].



(a) Standard Deployment (b) Our Deployment

Fig. 1. Schematic drawings of the architecture of database-driven hosted website in standard deployment and with our solution

3.1 Security of Database Credentials

A big security vulnerability in outsourced environments is the existence of clear-text database credentials and connection strings inside server-side script files. Existing common practice is to place this information in another script file that resides outside of the document root and is included during run-time [4, 19]. While this simple approach might protect the included file from a poorly configured web-server, it does nothing to prevent a user (or a superuser) on the machine from reading it.

To solve the clear-text database credential problem, our approach is to hide the database credentials by encrypting them, and storing them in a separate database, which we call *SafeDB*. Because our solution includes access to *SafeDB*, we need to protect ourselves from having that access information freely available. We achieve this in two steps. First, we compile the module that accesses the database and derive its database access password from a signed SHA-1 hash of the module's own executable. Second, this module checks if it was run by an authorized script file on a trusted web-server.

[7] The results of this process are the inputs for the run-time module and so may reside on the web-server.

Note that the location of this database may vary. Locating it locally with the web server may reduce the system's security, as a superuser may connect to and alter *SafeDB*. Locating it off the web server improves the security guarantees, while a distributed deployment may reduce the reliability and stability of the service. We further evaluate and analyze the performance in Section 5.

3.2 Key Generation and Solution Setup

During compile time, two sets of 2048-bit RSA keys are generated for the *Signer module* and *Run-Time Module* (*RTM*), which are described in the next section. The keys are placed in header files included by the *Signer* and *RTM*, respectively. The SHA-1 hashes of a valid web-server executable and related modules are compiled into the *RTM* as well as the owner's e-mail address. Once the package is compiled, one can begin incorporating our solution into the website's script files. A part of an unchanged script file (PHP) is shown in Listing 1, where the IP address, database user, database password and the name of the database are all in clear-text. To demonstrate the ease of using our solution, we show how to convert the legacy code in Listing 1 to safer code in Listing 2.

Listing 1. PHP Script connecting to a Database

```
<?php
...
$db = mysql_connect(192.168.0.100, 'my_usr', 'PWord');
mysql_select_db('my_store_db');
...
?>
```

Listing 2. PHP Script connecting to a database using SafeWS. With SafeWS, sensitive database information is not exposed.

```
<?php
...
$info = safe_exec('/home/u1/bin/rtm', 'tag_f11');
list($db_host, $db_name, $db_user, $db_pass) =
        split(':', $info, 4);
mysql_connect($db_host, $db_user, $db_pass);
mysql_select_db($db_name);
...
?>
```

3.3 Run-Time Module and Signer Module

The Run-Time Module (*RTM*) is the most crucial component in our solution. It is called within an authorized script file using *safe_exec()* (described below). When *RTM* loads, it calculates the SHA-1 hashes of its own executable, the script file that called it, as well as the executable of the web server (i. e., the calling process) and its relevant modules. When executing, *RTM* is supplied with a *tag* parameter. This piece of information allows one script to request many different sets of Database credentials from *RTM*. *RTM* then checks that it was called by a valid web server by comparing SHA-1 hashes.

RTM then attempts to access *SafeDB*. If the module was tampered with, it will not be able to derive the correct database password from the SHA-1 hash of its own executable.

Upon connecting to the database, *RTM* looks for a record that matches the SHA-1 hash of the script filename (that called *RTM*) with the supplied *tag*. Using its private key, *RTM* decrypts one of the fields and verifies the signature of the *Signer* module, which is described next. If the hash of the script file is verified, then it is authentic, and the database connection parameters as well as credentials are returned to the calling script file.

The *Signer* module contains its own private key, as well as the *RTM*'s public key. This module resides on the website owner's machine, which is assumed to be safe. Whenever a new script file is ready to be put on the website, the owner converts it to be compatible with *SafeWS*.

The website owner uses the *Signer* to sign the SHA-1 hash of a script file that will be installed on the server, and encrypts the result with the *RTM*'s public key. The *Signer* also associates a tag with that script file and with the set of credentials given by the owner.

The output of the *Signer* is a *cryptographic SQL file* that includes the following information: a SHA-1 hash of the script full path name with the tag, credentials and database connection parameters encrypted with the public key of the *RTM* and the SHA-1 hash of the script file signed using the *Signer*'s private key and encrypted by the *RTM*'s public key. This SQL file is transferred to the server containing the *SafeWS* database and is executed there.

3.4 PHP Limitation and *safe_exec()*

As we chose to evaluate our prototype with PHP scripts, we found a serious security limitation that affects PHP security and the security of PHP-based web hosting in general. In existing PHP execution environments, it is impossible to learn or verify the provenance (i.e., origin) of the caller.

PHP offers the following methods to execute non-PHP programs: *exec()*, *passthru()*, *proc_open()*, *popen()*, *shell_exec()* and *system()*. The process that is executed as a result of these calls does not know where the call originated (i.e. from which PHP file). Moreover, the parameters of the web-server session are not provided either. While it is possible to use *proc_open()* and pass environment variables to a new process, this would undermine the information integrity, as a hacker might try to pass bogus parameters to bypass our protection. This security limitation may or may not affect other scripting languages.

To solve these problems, we had to add another module to *SafeWS*'s architecture. We created a new run-time PHP module called *safe_exec()*. The main advantage of *safe_exec()* is that it is able to pass web-session information as well as run-time environment information, including the calling PHP file, into the process that it executes.

safe_exec() is integrated with our solution through *RTM*. This module needs to be installed on the service provider's machine, and added to the list of files *RTM* authenticates. A similar module may be needed for other scripting languages which do not pass credible execution and web environment information to executed binaries. CGI (Common Gateway Interface) binaries executed directly by Apache do receive the required information, and thus can pass it through *execle()*.

3.5 SafeWS Run-Time Protocol

Once *SafeWS* is deployed and configured, it will be invoked by the web-server when a participating script file is processed. *SafeWS*'s protocol is illustrated in Figure 2 and is described as follows.

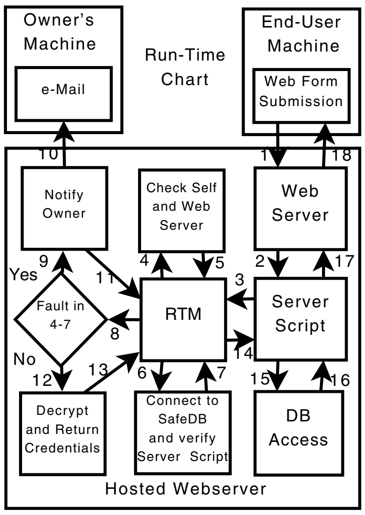


Fig. 2. SafeWS architecture and run-time information flow

– End-users submit an HTML form on their browser in Step 1, using either HTTP GET or HTTP POST.
– In Step 2, the hosted web-server executes the script that handles the data. The web-server then passes the information from the end-user to the script, through either environment variables or as the script's standard input. After initial processing, the script needs to set the database connection parameters, in order to process the end-user's request.
– The script calls *RTM* using *safe_exec* in Step 3. Please refer to Listing 2 for an example of such a call.
– Once *SafeWS*'s *RTM* module is started, it computes the SHA-1 hash of its caller's executable file (e.g. *apache*) in Step 4 and 5. It computes its own SHA-1 hash, which is then signed with *RTM*'s private key to produce the password for *SafeDB*.

- In Step 6 and 7, *RTM* attempts to access *SafeDB* with the credentials it was compiled with, as well as the password that it computed. *RTM* computes the SHA-1 hash of the caller PHP script, as well as a digest of the script's location and tag. The digest is used as a key to find the row in *SafeDB* that corresponds to the calling script. Upon selecting the relevant information from *SafeDB*, *RTM* attempts to verify the signed SHA-1 hash of the script. Namely, *RTM* (1) verifies the web-server executable and modules, (2) connects to *SafeDB*, (3) locates the correct record for the script, and (4) verifies the signature on its hash.
- Any failure in the above verification procedures results in *RTM* notifying the website owner of the security concern, as well as not returning the requested information to the calling script. This failure in turn would cause the script's subsequent connection to the database to fail, as shown in Step 9 through 11.
- In Step 12, upon a successful verification of the location and authenticity of the calling script, *RTM* decrypts the remainder of *SafeDB*'s record using its private key and obtains the address of the website's database server, as well as the database name, username and password required to access it. This information is then returned to the calling script in Step 13 and 14. The script then parses the data and connects to the website's database in Step 15 and 16, and can then complete its task.

In Figure 2, for the sake of description simplicity, we show *SafeDB* and the protected website's database all on the same local host as the web server. In practice, as we mentioned in Section 2, the database systems should preferably reside and be maintained at a different location from the local hosted web server, in order to reduce the security risk. Our protocol description and implementation can be directly used to accommodate such a distributed deployment of *SafeWS*.

3.6 Protecting against Advanced Hackers

As we stated in Section 2, advanced hackers may reverse engineer any program. This means that *RTM* would be vulnerable to attacks. If attackers reverse engineer *RTM* they could find the keys stored in it as well as the connection information to *SafeDB*. However, due to the hurdles we built into *SafeWS*, this process might take a considerable amount of time. First, *RTM* would have to be decompiled, the code (which may or may not resemble the original code) must be analyzed. The database credentials are made by using the SHA-1 hash of the *RTM* executable file signed by the private key stored in *RTM*. The attacker would need to build a program to use the extracted keys and obtain and sign the SHA-1 hash of *RTM*. In addition, the attackers must refrain from running or misusing *RTM* as the owners would be notified. Since we assume that *RTM* may be hacked, we can add another layer of security. By having a periodical job (such as a cron job) that would generate new keys, recompile both *RTM* and *Signer*, change the database credentials and distribute the new *RTM* to the server, we can reduce the probability of a successful attack. This periodical script would run on the trusted owner's machine, and would perform its duties every $X - 1$ seconds where X is the minimum number of seconds that an advanced hacker would take to obtain the *SafeDB* credentials.

3.7 RTM Design Aspects

RTM is executed each time a script with database access is run by the web server. Although this is sub-optimal performance wise, the security aspects were more important. By letting *RTM* be resident in memory, we could implement caching of database credentials, as well as avoid re-examining the web server executable and its modules (provided we can guarantee that the process is the same). This would improve the performance of *SafeWS* considerably, however, there are a couple of caveats to this approach which made us choose the other design. First, the system is designed for shared hosting environments, which normally do not allow their customers (website owners) to have resident services running on the machine. Second, the lack of a direct execution relationship between a script and *RTM* would reduce the knowledge the system gives *RTM* about the caller, as well as complicate calling *RTM* inside the scripts.

4 Discussion

In this section, we analyze the security properties and discuss practical considerations associated with deploying *SafeWS*. *SafeWS* satisfies the security requirements that are defined in Section 2 including the secrecy of database credentials, provenance authentication for database access, and integrity of outsourced environments, which are explained in detail next.

SafeWS guarantees that only authorized webpages from the web server can access a database that stores end user or product information. This property is achieved by storing the properties and environments of authorized webpages in *SafeDB*, which can only be accessed by *RTM* with the proper privately generated password. These operations correspond to Step 6 and 7 in Figure 2. In addition, our basic script run-time module *safe_exec()* ensures that only authentic information about script environments is passed to *RTM* for the verification purpose, and spoofing attacks (e.g., lying about an IP address or location) can be identified.

SafeWS effectively hides database access credentials from web server administrators who are allowed to read any file on the system. As stated earlier in Section 3, while using *SafeWS*, the website owner should separate the customer data from the running environment, i.e., scripts and data should reside on different servers. Sensitive data should be kept encrypted in a database, and the decrypting web server should be different than the encrypting one. With this separation of duties, even if attackers obtained the database credentials where the sensitive data is stored, they will not have a way to decrypt the encrypted customer data.

SafeWS ensures the integrity of outsourced websites by detecting and monitoring suspicious environments and activities and provides website owners with notifications of suspicious activities. The verification is realized mainly by our Run-Time Module in *SafeWS*. *RTM* leverages *safe_exec()*'s ability to pass web-session information as well as run-time environment information into the process that it executes. In the *SafeWS* run-time protocol, *RTM* checks the integrity of the web-server executable and verifies the signature on the hash of the invoked script against the correct record in *SafeDB* that can only be accessed by *RTM*.

Typically when server-side scripts are first developed, they are changed often due to programming errors or inappropriate specifications. But most of such scripts reach stable states and are rarely changed afterwards. Therefore, updates in *SafeWS* caused by script changes do not cause much communication and computation overheads. We give a more thorough evaluation and discussion on the performance of *SafeWS* in Section 5.

Note that in-memory code mutation or memory scanning are types of attacks that may find the private key or clear-text passwords [12]. Our current *SafeWS* design can withstand advanced reverse engineering attacks against *RTM* with the help of a periodically running script, as described in Seciont 3.6. However, we believe that these types of attacks would take significant efforts. Due to the fact that the *Signer* module is safely located on the website owner's private machine, even if the *RTM*'s private key is recovered, the attacker will not be able to sign altered or new script files without the owner's key, and thus will be unable to hijack the website without being detected.

5 Evaluation

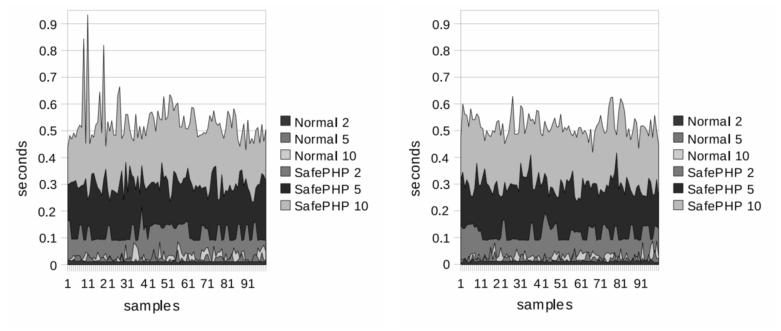
Our goals were to see whether *SafeWS* was a viable solution for small and medium third-party hosted web-sites. We decided to check the impact on both end-users and web server machines. As end-users today expect fast response times, we wanted to achive sub-second end-to-end times and low server impact.

5.1 Experiment Setup

We used two servers for testing *SafeWS*. Our web server machine is an Intel dual core (2*1.6Ghz) with 1MB cache and 2GB RAM. This machine runs Linux kernel version 2.6.23.17-88 SMP, Apache 2.2.8 and MySQL 5.0.45. Our client emulator and remote database machine is an Intel dual core (2*2.8Ghz), with 1MB cache and 2GB RAM running Linux version 2.6.27.9 SMP and MySQL 5.0.67. We picked two common website procedures that may allow access to sensitive information. We wrote PHP scripts that store and update a database with this information. We measured the end-to-end performance of those functions, then converted the PHP scripts to work with *SafeWS* and re-measured. Each of the following experiments were conducted using both local (on the web server machine) as well as remote (on the client emulator machine) databases.

Addition of Users. We created a PHP script that processes an 'HTTP GET' form which creates a new web-site user. There were seven pieces of information stored for each user: First name, Last name, Address, City, Zip code, e-mail address and password. We then ran a program that produced random values for these fields and ran a multi-threaded program that generated varying amounts of concurrent sessions. We measured each session from start to finish.

Changing Website Passwords. We created another PHP script that handles changes to a website-user's password. The fields provided were the users e-mail address, the old password, and the new password. Using the stored generated data from the previous procedure, we generated new passwords and ran our session generator with varying amounts of concurrent sessions and measured their end-to-end performance.



(a) Addition of Users (b) Changing Website Passwords

Fig. 3. Concurrency vs. Response Times with and without SafeWS. Normal refers to non-SafeWS measurements, numbers refer to concurrent sessions.

5.2 Experiment Results

We were able to achieve a sustainable peak performance of over 72,000 user addition and password changing requests per hour. Although the web server running the RTM had a load average of 12, the average end-user experience was under 0.5 seconds (from browser connection to the web server until the response was fully available). Using slightly less concurrency (5 simultaneous requests at all times) we achieved a sustainable average of 0.3 seconds end-user response time and a server load average of 1.4. This translates to 57,600 requests per hour or as much as 1.38 million requests per day (for uniform distribution of visits). Our measurements showed no significant performance difference between local vs. remote database use. We believe this is due to us using a machine on the same LAN, as well as a reduction in resource consumption. Although the performance overhead of *SafeWS* is significantly over the average running time of the scripts that do not use *SafeWS* (refer to Figure 3), the end-to-end performance is still sub-second, and can be improved further by optimizing the web server and database server software.

6 Related Work

With the increasing development of IT outsourcing, a substantial amount of research work has been done on how to verify outsourced data and computation [2, 3, 9–11, 15–17]. Merkle hash trees have been used extensively for authentication of data elements [14]. Aggregate signatures are another approach for data authentication, where each data tuple is signed by the data owner [17]. Most recently, the privacy issue in verifying queries was first addressed by in [18] which gave an elegant solution using hashing for proving the completeness of selection queries without revealing neighboring entries.

Database-as-a-service (DAS) model [10, 11, 16, 16] is an instantiation of the computing model involving trusted clients, who store their data at an untrusted server that is administrated by a service provider. The challenge in DAS is to make it impossible for the service provider to correctly interpret the data. The data is owned by clients. The clients only have limited computational power and storage, and they rely on the server for the mass computational power and storage. Hacigümüs, Iyer, and Mehrotra [11] addressed the execution of aggregate queries over encrypted data using homomorphic encryption scheme. Mykletun and Tsudik [16] proposed an alternative approach where the data owner pre-computes and encrypts the aggregate results and stores them at the service provider. This approach avoids the use of homomorphic encryption, which was found to have a security flaw when used for DAS [16]. Our model is different from DAS, and is suitable for a more general security setting, as the data does not have to originate from the client.

Efforts to discern the trustworthiness of a server (and in some cases alert web users to untrusted servers) utilizing hardware such as the Trusted Platform Module (TPM) [8] and using commitments and attestations [1, 13] and their combinations [20] have been made. However, these solutions do not protect against obtaining database credentials from a text file, and also require specialized hardware and kernel modification on the web-server side, as well as software on the client side, and trusted authorities to provide verification. In comparison, *SafeWS* is easy to adopt and more efficient.

7 Conclusions and Future Work

Outsourced information is as safe as the security provided by the server storing it. In order to improve the security of outsourced websites, we presented *SafeWS* in this paper. *SafeWS* is a protocol encompassing a distributed architecture that provides a robust layer of security between web server-side scripts and databases, while notifying site owners of anomalous run-time behavior. We gave the security models and definitions associated with *SafeWS* in the outsourced web service scenario. We implemented *SafeWS* system in C/C++ and performed extensive experimental evaluation on the performance and robustness of the system. Our results showed that the security overhead introduced by *SafeWS* is low at the web server side even when the number of users is large.

For future work, we plan to leverage the infrastructure provided by *SafeWS* to extend the protection to cross-site scripting (XSS). One promising approach is to add the identifiers of allowed referer pages into the *SafeWS* database, the same way as we retrieve, store, and verify this information from the web server. We also plan to further improve the performance and robustness of the *SafeWS* implementation.

Dedication

The authors would like to dedicate this paper in memory of Denitsa Tilkidjieva. A dear friend and a bright third-year Ph.D. student at the Computer Science department in Rutgers. She passed away January 22nd, 2009 and will always be missed.

References

1. Arbaugh, W.A., Farber, D.J., Smith, J.M.: A secure and reliable bootstrap architecture. In: Proceedings of the 1997 IEEE Symposium on Security and Privacy, pp. 65–71. IEEE Computer Society, Los Alamitos (1997)

2. Bertino, E., Ooi, B.C., Yang, Y., Deng, R.H.: Privacy and ownership preserving of outsourced medical data. In: Proceedings of the 21st International Conference on Data Engineering (ICDE), pp. 521–532 (2005)
3. Devanbu, P., Gertz, M., Martel, C., Stubblebine, S.: Authentic third-party data publication. Journal of Computer Security 11(3) (2003)
4. Dickinson, P.: Top 7 PHP Security Blunders (December 2005), `http://www.sitepoint.com/article/php-security-blunders/`
5. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard) (August 2008), `http://www.ietf.org/rfc/rfc5246.txt`
6. Foster, I., Kesselman, C., Nick, J.M., Tuecke, S.: Grid services for distributed system integration. Computer 35(6), 37–46 (2002)
7. GoDaddy.com. Why You Need An SSL Certificate, `https://www.godaddy.com/gdshop/pdf/SSLMarketingGuideGodaddy.pdf`
8. Trusted Computing Group. TCG 1.2 specifications, `https://www.trustedcomputinggroup.org/`
9. Hacigümüs, H., Iyer, B., Li, C., Mehrotra, S.: Executing SQL over encrypted data in the database-service provider model. In: Proceedings of ACM SIGMOD Conference on Management of Data, pp. 216–227. ACM Press, New York (2002)
10. Hacigümüs, H.B., Iyer, H.B., Mehrotra, S.: Providing database as a service. In: Proceedings of International Conference on Data Engineering (ICDE) (March 2002)
11. Hacigümüs, H., Iyer, B., Mehrotra, S.: Efficient execution of aggregation queries over encrypted relational databases. In: Lee, Y., Li, J., Whang, K.-Y., Lee, D. (eds.) DASFAA 2004. LNCS, vol. 2973, pp. 125–136. Springer, Heidelberg (2004)
12. Halderman, J.A., Schoen, S.D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J.A., Feldman, A.J., Appelbaum, J., Felten, E.W.: Lest we remember: Cold boot attacks on encryption keys. In: van Oorschot, P.C. (ed.) USENIX Security Symposium, pp. 45–60. USENIX Association (2008)
13. Lampson, B., Burrows, M., Wobber, E.: Authentication in distributed systems: Theory and practice. ACM Transactions on Computer Systems 10, 265–310 (1992)
14. Merkle, R.: Protocols for public key cryptosystems. In: Proceedings of the 1980 Symposium on Security and Privacy, pp. 122–133. IEEE Computer Society Press, Los Alamitos (1980)
15. Mykletun, E., Narasimha, M., Tsudik, G.: Authentication and integrity in outsourced databases. In: Proceedings of Symposium on Network and Distributed Systems Security (NDSS) (February 2004)
16. Mykletun, E., Tsudik, G.: Aggregation queries in the database-as-a-service model. In: Damiani, E., Liu, P. (eds.) Data and Applications Security 2006. LNCS, vol. 4127, pp. 89–103. Springer, Heidelberg (2006)
17. Narasimha, M., Tsudik, G.: Authentication of outsourced databases using signature aggregation and chaining. In: Li Lee, M., Tan, K.-L., Wuwongse, V. (eds.) DASFAA 2006. LNCS, vol. 3882, pp. 420–436. Springer, Heidelberg (2006)
18. Pang, H., Jain, A., Ramamritham, K., Tan, K.-L.: Verifying completeness of relational query results in data publishing. In: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pp. 407–418 (2005)
19. Shiflett, C.: Security corner: Shared hosting. *php—architect* 3(3) (March 2004), `http://shiflett.org/articles/shared-hosting`
20. Xu, G., Borcea, C., Iftode, L.: Satem: Trusted service code execution across transactions. In: IEEE Symposium on Reliable Distributed Systems, pp. 321–336 (2006)