SQLrand: Preventing SQL Injection Attacks

Stephen W. Boyd and Angelos D. Keromytis

Department of Computer Science
Columbia University
{*swb48,angelos*}*@cs.columbia.edu*

Abstract. We present a practical protection mechanism against SQL injection attacks. Such attacks target databases that are accessible through a web front-end, and take advantage of flaws in the input validation logic of Web components such as CGI scripts. We apply the concept of instruction-set randomization to SQL, creating instances of the language that are unpredictable to the attacker. Queries injected by the attacker will be caught and terminated by the database parser. We show how to use this technique with the MySQL database using an intermediary proxy that translates the random SQL to its standard language. Our mechanism imposes negligible performance overhead to query processing and can be easily retrofitted to existing systems.

1 Introduction

The prevalence of buffer overflow attacks [3, 29] as an intrusion mechanism has resulted in considerable research focused on the problem of preventing [14, 11], detecting [35, 23, 25], or containing [33, 31, 21, 12] such attacks. Considerably less attention has been paid to a related problem, SQL injection attacks [1]. Such attacks have been used to extract customer and order information from e-commerce databases, or bypass security mechanisms.

The intuition behind such attacks is that pre-defined logical expressions within a pre-defined query can be altered simply by injecting operations that always result in true or false statements. This injection typically occurs through a web form and associated CGI script that does not perform appropriate input validation. These types of injections are not limited strictly to character fields. Similar alterations to the "where" and "having" SQL clauses have been exposed, when the application does not restrict numeric data for numeric fields.

Standard SQL error messages returned by a database can also assist the attacker. In situations where the attacker has no knowledge of the underlying SQL query or the contributing tables, forcing an exception may reveal more details about the table or its field names and types. This technique has been shown to be quite effective in practice [5, 27].

One solution to the problem is to improve programming techniques. Common practices include escaping single quotes, limiting the input character length, and filtering the exception messages. Despite these suggestions, vulnerabilities continue to surface in web applications, implying the need for a different approach. Another approach is to use the PREPARE statement feature supported by many databases, which allows a client

to pre-issue a template SQL query at the beginning of a session; for the actual queries, the client only needs to specify the variables that change. Although the PREPARE feature was introduced as a performance optimization, it can address SQL injection attacks if the same query is issued many times. When the queries are dynamically constructed (*e.g.,* as a result of a page with several options that user may select), this approach does not work as well.

[22] introduced the concept of instruction-set randomization for safeguarding systems against any type of code-injection attack, by creating process-specific randomized instruction sets (*e.g.,* machine instructions) of the system executing potentially vulnerable software. An attacker that does not know the key to the randomization algorithm will inject code that is invalid for that randomized processor (and process), causing a runtime exception.

We apply the same technique to the problem of SQL injection attacks: we create randomized instances of the SQL query language, by randomizing the template query inside the CGI script and the database parser. To allow for easy retrofitting of our solution to existing systems, we introduce a de-randomizing proxy, which converts randomized queries to proper SQL queries for the database. Code injected by the rogue client evaluates to undefined keywords and expressions. When this is the outcome, then standard keywords (*e.g.,* "or") lose their significance, and attacks are frustrated before they can even commence. The performance overhead of our approach is minimal, adding up to 6.5*ms* to query processing time.

We explain the intuition behind our system, named *SQLrand*, in Section 2, and describe our prototype implementation in Section 3. We give some performance results in Section 4, and an overview of related work in Section 5.

2 SQLrand System Architecture

Injecting SQL code into a web application requires little effort by those who understand both the semantics of the SQL language and CGI scripts. Numerous applications take user input and feed it into a pre-defined query. The query is then handed to the database for execution. Unless developers properly design their application code to protect against unexpected data input by users, alteration to the database structure, corruption of data or revelation of private and confidential information may be granted inadvertently.

For example, consider a login page of a CGI application that expects a user-name and the corresponding password. When the credentials are submitted, they are inserted within a query template such as the following:

```
"select * from mysql.user
   where username=' " . $uid . " ' and
        password=password(' ". $pwd . " ');"
```

Instead of a valid user-name, the malicious user sets the $uid variable to the string: ' or 1=1; - -', causing the CGI script to issue the following SQL query to the database:

```
"select * from mysql.user
```

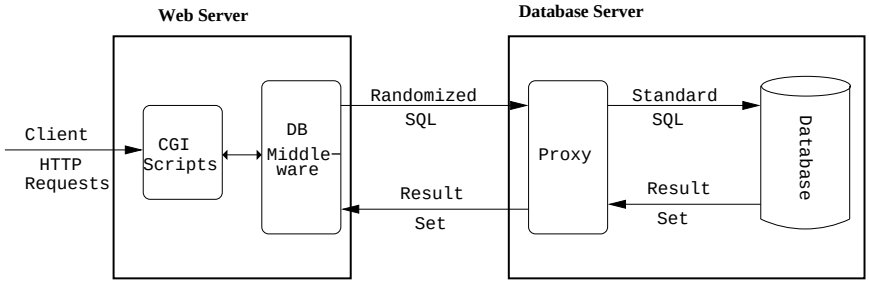

Fig. 1. SQLrand System Architecture

```
where username='' or 1=1; - -'' and
      password=password('_any_text_');"
```

Notice that the single quotes balance the quotes in the pre-defined query, and the double hyphen comments out the remainder of the SQL query. Therefore, the password value is irrelevant and may be set to any character string. The result set of the query contains at least one record, since the "where" clause evaluates to true. If the application identifies a valid user by testing whether the result set is non-empty, the attacker can bypass the security check.

Our solution extends the application of Instruction-Set Randomization [22] to the SQL language: the SQL standard keywords are manipulated by appending a random integer to them, one that an attacker cannot easily guess. Therefore, any malicious user attempting an SQL injection attack would be thwarted, for the user input inserted into the "randomized" query would always be classified as a set of non-keywords, resulting in an invalid expression.

Essentially, the structured query language has taken on new keywords that will not be recognized by the database's SQL interpreter. A difficult approach would be to modify the database's interpreter to accept the new set of keywords. However, attempting to change its behavior would be a daunting task. Furthermore, a modified database would require all applications submitting SQL queries to conform to its new language. Although dedicating the database server for selected applications might be possible, the random key would not be varied among the SQL applications using it. Ideally, having the ability to vary the random SQL key, while maintaining one database system, grants a greater level of security, by making it difficult to subvert multiple applications by successfully attacking the least protected one.

Our design consists of a proxy that sits between the client and database server (see Figure 1). Note that the proxy may be on a separate machine, unlike the figure's depiction.

By moving the de-randomization process outside the DataBase Management System (DBMS) to the proxy, we gain in flexibility, simplicity, and security. Multiple proxies using unique random keys to decode SQL commands can be listening for connections on behalf of the same database, yet allowing disparate SQL applications to communicate in their own "tongue." The interpreter is no longer bound to the internals of

the DBMS. The proxy's primary obligation is to decipher the random SQL query and then forward the SQL command with the standard set of keywords to the database for computation. Another benefit of the proxy is the concealment of database errors which may unveil the random SQL keyword extension to the user. A typical attack consists of a simple injection of SQL, hoping that the error message will disclose a subset of the query or table information, which may be used to deduce intuitively hidden properties of the database. By stripping away the randomization tags in the proxy, we need not worry about the DBMS inadvertently exposing such information through error messages; the DBMS itself never sees the randomization tags. Thus, to ensure the security of the scheme, we only need to ensure that no messages generated by the proxy itself are ever sent to the DBMS or the front-end server. Given that the proxy itself is fairly simple, it seems possible to secure it against attacks. In the event that the proxy is compromised, the database remains safe, assuming that other security measures are in place.

To assist the developer in randomizing his SQL statements, we provide a tool that reads an SQL statement(s) and rewrites all keywords with the random key appended. For example, in the C language, an SQL query, which takes user input, may look like the following:

```
select gender, avg(age)
   from cs101.students
      where dept = %d
   group by gender
```

The utility will identify the six keywords in the example query and append the key to each one (*e.g.*, when the key is "123"):

```
select123 gender, avg123 (age)
   from123 cs101.students
      where123 dept = %d
   group123 by123 gender
```

This SQL template query can be inserted into the developer's web application. The proxy, upon receiving the randomized SQL, translates and validates it before forwarding it to the database. Note that the proxy performs simple syntactic validation — it is otherwise unaware of the semantics of the query itself.

3 Implementation

To determine the practicality of the approach we just outlined, we built a proof-of-concept proxy server that sits between the client (web server) and SQL server, de-randomizes requests received from the client, and conveys the query to the server. If an SQL injection attack has occurred, the proxy's parser will fail to recognize the randomized query and will reject it. The two primary components were the de-randomization element and the communication protocol between the client and database system. In order to de-randomize the SQL query, the proxy required a modified SQL parser that

expected the suffix of integers applied to all keywords. As a "middle man," it had to conceal its identity by masquerading as the database to the client and vice versa. Although our implementation focused on CGI scripts as the query generators, a similar approach applies when using JDBC.

The randomized SQL parser utilized two popular tools for writing compilers and parsers: flex and yacc. Capturing the encoded tokens required regular expressions that matched each SQL keyword (case-insensitive) followed by zero or more digits. (Technically, it did not require a key; practically, it needs one.) If properly encoded, the lexical analyzer strips the token's extension and returns it to the grammar for reassembly with the rest of the query. Otherwise, the token remains unaltered and is labeled as an identifier. By default, flex reads a source file, but our design required an array of characters as input. To override this behavior, the YY_INPUT macro was re-defined to retrieve tokens from a character string introduced by the proxy. During the parsing phase, any syntax error signals the improper construction of an SQL query using the pre-selected random key. Either the developer's SQL template is incorrect or the user's input includes unexpected data, whether good or bad. On encountering this, the parser returns NULL; otherwise, in the case of a successful parse, the de-randomized SQL string is returned. The parser was designed as a C library.

With the parser completed, the communication protocol had to be established between the proxy and a database. We used MySQL, a popular and widely used open-source database system, to create a fictitious customer database. The record size of the tables ranged from twenty to a little more than eleven thousand records. These sample tables were used in the evaluation of benchmark measurements described in Section 4. The remaining piece involved integrating the database's communication mechanism within the proxy.

Depending upon the client's language of choice, MySQL provides many APIs to access the database, yet the same application protocol. Since the proxy will act as a client to the database, the C API library was suitable. One problem existed: the mysqlclient C library does not have a server-side counterpart for accepting and disassembling the MySQL packets sent using the client API. Therefore, the protocol of MySQL had to be analyzed and incorporated into the proxy. Unfortunately, there was no official documentation; however, a rough sketch of the protocol existed which satisfied the requirements of the three primary packets: the query, the error, and the disconnect packets.

The query packet carries the actual request to the database. The quit message is necessary in cases where the client is abruptly disconnected from the proxy or sends an invalid query to the proxy. In either case the proxy gains the responsibility of discretely disconnecting from the database by issuing the quit command on behalf of the client. Finally, the error packet is only sent to the client when an improper query generates a syntax error, thus indicating a possible injection attack.

The client application needs only to define its server connection to redirect its packets through the proxy rather than directly to the database. In its connection method, this is achieved simply by changing the port number of the database to the port where the proxy is listening. After receiving a connection, the proxy in turn establishes a connection with the database and hands off all messages it receives from the client. If the command byte of the MySQL packet from the client indicates the packet contains a

query, the proxy extracts the SQL and passes it to the interpreter for decoding. When unsuccessful, the proxy sends an error packet with a generic "syntax error" message to the client and disconnects from the database. On the other hand, a successful parsing of the SQL query produces a translation to the de-randomized syntax. The proxy overwrites the original, randomized query with the standard query that the database is expecting into the body of the MySQL packet. The packet size is updated in the header and pushed out to the database. The normal flow of packets continues until the client requests another query.

The API libraries define some methods which will not work with the proxy, as they hardcode the SQL query submitted to the database. For example, mysql_list_dbs() sends the query "SHOW databases LIKE <wild-card-input>". Without modification to the client library, the workaround would be to construct the query string with the proper randomized key and issue the mysql_query() method. Presently, binary SQL cannot be passed to the proxy for processing; therefore, mysql_real_query() must be avoided.

4 Evaluation

To address the practicality of using a proxy to de-randomize encoded SQL for a database, two objectives were considered. First, the proxy must prevent known SQL injection vulnerabilities within an application. Second, the extra overhead introduced by the proxy must be evaluated.

4.1 Qualitative Evaluation

First, a sample CGI application was written, which allowed a user to inject SQL into a "where" clause that expected an account ID. With no input validation, a user can easily inject SQL to retrieve account information concerning all accounts. When using the SQLrand proxy, the injected statement is identified and an error message issued, rather than proceeding with the processing of the corrupted SQL query. After testing the reliability of the proxy on a "home grown" example, the next step was to identify an SQL injection vulnerability in a pre-existing application.

An open-source bulletin board, phpBB v2.0.5, presented an opportunity to inject SQL into viewtopic.php, revealing the password of a user one byte at a time. After the attack was replicated in the test environment, the section of vulnerable SQL was randomized and the connection was redirected through the proxy. As expected, the proxy recognized the injection as invalid SQL code and did not send it to the database. The phpBB application did not succumb to the SQL injection attack as verified without the proxy. However, it was observed that the application displays an SQL query to the user by default when zero records are returned. Since an exception does not return any rows, the proxy's encoding key was revealed. Again, the randomization method still requires good coding practices. If a developer chooses to reveal the SQL under certain cases, there is little benefit to the randomization process. Of course, one must remember that the application was not designed with the proxy implementation in mind.

Another content management application prone to SQL injection attacks, Php-Nuke depends on the magic_quotes_gpc option to be turned on to protect against some of

them. Without this setting, several modules are open to such attacks. Even with the option set, injections on numeric fields are not protected because the application does not check for numeric input. For example, when attempting to download content from the php-nuke application, the download option d_op is set to 'getit' and accepts an unchecked, numeric parameter name 'lid'. It looks up the URL for the content from the download table based on the lid value and sets it in the HTTP location header statement. If an attacker finds an invalid lid (determined by PHP-Nuke reloading its home page) and appends 'union select pass from users_table' to it, the browser responds with an error message stating that the URL had failed to load, thus revealing the sensitive information. However, when applying the proxy, injection attacks in the affected download module were averted. These vulnerabilities are open in other modules within PHP-Nuke that would also be quickly secured by using the proxy. The same common injection schemes are cited in various applications.

4.2 Performance Evaluation

Next, we quantified the overhead imposed by SQLrand. An experiment was designed to measure the additional processing time required by three sets of concurrent users, respectively 10, 25, and 50. Each class executed, in a round-robin fashion, a set of five queries concurrently over 100 trials. The average length of the five different queries was 639 bytes, and the random key length was thirty-two bytes. The sample customer database created during the implementation was the target of the queries. The database, proxy, and client program were on separate *x86* machines running RedHat Linux, within the same network. The overhead of proxy processing ranged from 183 to 316 microseconds for 10 to 50 concurrent users respectively. Table 1 shows the proxy's performance.

Table 1. Proxy Overhead (in microseconds)

Users	Min	Max	Mean	Std
10	74	1300	183.5	126.9
25	73	2782	223.8	268.1
50	73	6533	316.6	548.8

The worst-case scenario adds approximately 6.5 milliseconds to the processing time of each query. Since acceptable response times for most web applications usually fall between a few seconds to tens of seconds, depending on the purpose of the application, the additional processing time of the proxy contributes insignificant overhead in a majority of cases.

5 Related Work

To date, little attention has been paid to SQL injection attacks. The work conceptually closest to ours is RISE [8], which applies a randomization technique similar to our

Instruction-Set Randomization [22] for binary code only, and uses an emulator attached to specific processes. The inherent use of and dependency on emulation makes RISE simultaneously more practical for immediate use and inherently slower in the absence of hardware. [26] uses more general code obfuscation techniques to harden program binaries against static disassembly.

In the general area of randomization, originally proposed as a way of introducing diversity in computer systems [16], notable systems include PointGuard and Address Obfuscation. PointGuard [11] encrypts all pointers while they reside in memory and decrypts them only before they are loaded to a CPU register. This is implemented as an extension to the GCC compiler, which injects the necessary instructions at compilation time, allowing a pure-software implementation of the scheme. Another approach, address obfuscation [10], randomizes the absolute locations of all code and data, as well as the distances between different data items. Several transformations are used, such as randomizing the base addresses of memory regions (stack, heap, dynamically-linked libraries, routines, static data, *etc.*), permuting the order of variables/routines, and introducing random gaps between objects (*e.g.,* randomly pad stack frames or *malloc()*'ed regions). Although very effective against *jump-into-libc* attacks, it is less so against other common attacks, since the amount of possible randomization is relatively small (especially when compared to our key sizes). However, address obfuscation can protect against attacks that aim to corrupt variables or other data. This approach can be effectively combined with instruction randomization to offer comprehensive protection against all memory-corrupting attacks. [13] gives an overview of various protection mechanisms, including randomization techniques, and makes recommendations on choosing obfuscation (of interface or implementation) *vs.* restricting the same.

[6] describes some design principles for safe interpreters, with a focus on JavaScript. The Perl interpreter can be run in a mode that implements some of these principles (access to external interfaces, namespace management, *etc.*). While this approach can somewhat mitigate the effects of an attack, it cannot altogether prevent, or even contain it in certain cases (*e.g.,* in the case of a Perl CGI script that generates an SQL query to the back-end database).

Increasingly, source code analysis techniques are brought to bear on the problem of detecting potential code vulnerabilities. The most simple approach has been that of the compiler warning on the use of certain unsafe functions, *e.g., gets()*. More recent approaches [17, 35, 23, 34, 15] have focused on detecting specific types of problems, rather than try to solve the general "bad code" issue, with considerable success. While such tools can greatly help programmers ensure the safety of their code, especially when used in conjunction with other protection techniques, they (as well as dynamic analysis tools such as [25, 24]) offer incomplete protection, as they can only protect against and detect *known* classes of attacks and vulnerabilities. Unfortunately, none of these systems have been applied to the case of SQL injection attacks.

Process sandboxing [31] is perhaps the best understood and widely researched area of containing bad code (or its effects), as evidenced by the plethora of available systems like Janus [21], Consh [4], Mapbox [2], OpenBSD's *systrace* [33], and the Mediating Connectors [7]. These operate at user level and confine applications by filtering access to system calls. To accomplish this, they rely on *ptrace(2)*, the */proc* file system, and/or

special shared libraries. Another category of systems, such as Tron [9], SubDomain [12] and others [18, 20, 36, 30, 37, 28, 32], go a step further. They intercept system calls inside the kernel, and use policy engines to decide whether to permit the call or not. The main problem with all these is that the attack is not prevented: rather, the system tries to limit the damage such code can do, such as obtain super-user privileges. In the context of a web server, this means that a web server may only be able to issue queries to particular databases or access a limited set of files, *etc.* [19] identifies several common security-related problems with such systems, such as their susceptibility to various types of race conditions.

6 Conclusions

We presented SQLrand, a system for preventing SQL injection attacks against web servers. The main intuition is that by using a randomized SQL query language, specific to a particular CGI application, it is possible to detect and abort queries that include injected code. By using a proxy for the de-randomization process, we achieve portability and security gains: the same proxy can be used with various DBMS back-end, and it can ensure that no information that would expose the randomization process can leak from the database itself. Naturally, care must be taken by the CGI implementor to avoid exposing randomized queries (as is occasionally done in the case of errors). We showed that this approach does not sacrifice performance: the latency overhead imposed on each query was at most 6.5 milliseconds.

We believe that SQLrand is a very practical system that solves a problem heretofore ignored, in preference to the more "high profile" buffer overflow attacks. Our plans for future work include developing tools that will further assist programmers in using SQLrand and extending coverage to other DBMS back-ends.

References

1. CERT Vulnerability Note VU#282403. `http://www.kb.cert.org/vuls/id/282403`, September 2002.
2. A. Acharya and M. Raje. Mapbox: Using parameterized behavior classes to confine applications. In *Proceedings of the 9th USENIX Security Symposium*, pages 1–17, August 2000.
3. Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), 1996.
4. A. Alexandrov, P. Kmiec, and K. Schauser. Consh: A confined execution environment for internet computations, December 1998.
5. C. Anley. Advanced SQL Injection In SQL Server Applications. `http://www.nextgenss.com/papers/advanced\_sql\_injection.pdf`, 2002.
6. V. Anupam and A. Mayer. Security of Web Browser Scripting Languages: Vulnerabilities, Attacks, and Remedies. In *Proceedings of the 7th USENIX Security Symposium*, pages 187–200, January 1998.
7. R. Balzer and N. Goldman. Mediating connectors: A non-bypassable process wrapping technology. In *Proceeding of the 19th IEEE International Conference on Distributed Computing Systems*, June 1999.
8. E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *Proceedings*

of the 10th ACM Conference on Computer and Communications Security (CCS), pages 281–289, October 2003.

9. A. Berman, V. Bourassa, and E. Selberg. TRON: Process-Specific File Protection for the UNIX Operating System. In *Proceedings of the USENIX Technical Conference*, January 1995.
10. S. Bhatkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, August 2003.
11. C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting Pointers From Buffer Overflow Vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, pages 91–104, August 2003.
12. C. Cowan, S. Beattie, C. Pu, P. Wagle, and V. Gligor. SubDomain: Parsimonious Security for Server Appliances. In *Proceedings of the 14th USENIX System Administration Conference (LISA 2000)*, March 2000.
13. C. Cowan, H. Hinton, C. Pu, and J. Walpole. The Cracker Patch Choice: An Analysis of Post Hoc Security Techniques. In *Proceedings of the National Information Systems Security Conference (NISSC)*, October 2000.
14. C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, Jan. 1998.
15. N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, June 2003.
16. S. Forrest, A. Somayaji, and D. Ackley. Building Diverse Computer Systems. In *HotOS-VI*, 1997.
17. J. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, May 1999.
18. T. Fraser, L. Badger, and M. Feldman. Hardening COTS Software with Generic Software Wrappers. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1999.
19. T. Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proceedings of the Symposium on Network and Distributed Systems Security (SNDSS)*, pages 163–176, February 2003.
20. D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. E. Anderson. SLIC: An Extensibility System for Commodity Operating Systems. In *Proceedings of the 1998 USENIX Annual Technical Conference*, pages 39–52, June 1998.
21. I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Procedings of the 1996 USENIX Annual Technical Conference*, 1996.
22. G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proceedings of the ACM Computer and Communications Security (CCS) Conference*, pages 272–280, October 2003.
23. D. Larochelle and D. Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, pages 177–190, August 2001.
24. E. Larson and T. Austin. High Coverage Detection of Input-Related Security Faults. In *Proceedings of the 12th USENIX Security Symposium*, pages 121–136, August 2003.
25. K. Lhee and S. J. Chapin. Type-assisted dynamic buffer overflow detection. In *Proceedings of the 11th USENIX Security Symposium*, pages 81–90, August 2002.

26. C. Linn and S. Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, pages 290–299, October 2003.
27. D. Litchfield. Web Application Disassembly wth ODBC Error Messages. `http://www.nextgenss.com/papers/webappdis.doc`.
28. P. Loscocco and S. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the USENIX Annual Technical Conference, Freenix Track*, pages 29–40, June 2001.
29. M. Conover and w00w00 Security Team. w00w00 on heap overflows. `http://www.w00w00.org/files/articles/heaptut.txt`, January 1999.
30. T. Mitchem, R. Lu, and R. O'Brien. Using Kernel Hypervisors to Secure Applications. In *Proceedings of the Annual Computer Security Applications Conference*, December 1997.
31. D. S. Peterson, M. Bishop, and R. Pandey. A Flexible Containment Mechanism for Executing Untrusted Code. In *Proceedings of the 11th USENIX Security Symposium*, pages 207–225, August 2002.
32. V. Prevelakis and D. Spinellis. Sandboxing Applications. In *Proceedings of the USENIX Technical Annual Conference, Freenix Track*, pages 119–126, June 2001.
33. N. Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium*, pages 257–272, August 2003.
34. U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, pages 201–216, August 2001.
35. D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A First Step towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proceedings of the ISOC Symposium on Network and Distributed System Security (SNDSS)*, pages 3–17, February 2000.
36. K. M. Walker, D. F. Stern, L. Badger, K. A. Oosendorp, M. J. Petkac, and D. L. Sherman. Confining root programs with domain and type enforcement. In *Proceedings of the USENIX Security Symposium*, pages 21–36, July 1996.
37. R. N. M. Watson. TrustedBSD: Adding Trusted Operating System Features to FreeBSD. In *Proceedings of the USENIX Annual Technical Conference, Freenix Track*, pages 15–28, June 2001.