DOI:10.7544/issn1000-1239.2016.20160443
53(10):2262-2276,2016

基于敏感字符的SQL注入攻击防御方法

张慧琳[1] 丁 羽[1] 张利华[1] 段 镭[1] 张 超[2] 韦 韬[3] 李冠成[1] 韩心慧[1]
[1](北京大学计算机科学技术研究所 北京 100080)
[2](加州大学伯克利分校 加利福尼亚伯克利 94720)
[3](百度美国有限责任公司 加利福尼亚森尼韦尔 94089)
(zhanghuilin@pku.edu.cn)

SQL Injection Prevention Based on Sensitive Characters

Zhang Huilin[1], Ding Yu[1], Zhang Lihua[1], Duan Lei[1], Zhang Chao[2], Wei Tao[3], Li Guancheng[1], and Han Xinhui[1]
[1](*Institute of Computer Science and Technology, Peking University, Beijing* 100080)
[2](*University of California at Berkeley, Berkeley, CA*, 94720)
[3](*Baidu USA Limited Liability Company, Sunnyvale, CA*, 94089)

Abstract SQL injection attacks are prevalent Web threats. Researchers have proposed many taint analysis solutions to defeat this type of attacks, but few are efficient and practical to deploy. In this paper, we propose a practical and accurate SQL injection prevention method by tainting trusted sensitive characters into extended UTF-8 encodings. Unlike typical positive taint analysis solutions that taint all characters in hard-coded strings written by the developer, we only taint the trusted sensitive characters in these hard-coded strings. Furthermore, rather than modifying Web application interpreter to track taint information in extra memories, we encode the taint metadata into the bytes of trusted sensitive characters, by utilizing the characteristics of UTF-8 encoding. Lastly, we identify and escape untrusted sensitive characters in SQL statements to prevent SQL injection attacks, without parsing the SQL statements. A prototype called PHPGate is implemented as an extension on the PHP Zend engine. The evaluation results show that PHPGate can protect Web applications from real world SQL injection attacks and introduce a low performance overhead (less than 1.6%).

Key words SQL injection attack; trusted sensitive character; dynamic taint analysis; positive taint analysis; UTF-8 encoding

摘 要 SQL注入攻击历史悠久,其检测机制也研究甚广.现有的研究利用污点分析(taint analysis)结合SQL语句语法分析进行SQL注入攻击检测,但由于需要修改Web应用程序执行引擎来标记和跟踪污点信息,难以部署,并且时间和空间性能损失过大.通过分析SQL注入攻击机理,提出一种基于敏感字符的SQL注入攻击防御方法.1)仅对来自常量字符串的可信敏感字符进行积极污点标记;2)无需修改Web应用程序执行引擎,利用编码转换将污点信息直接存储在可信敏感字符的编码值中,动态跟踪其在程序中的传播;3)无需SQL语句语法分析,只需利用编码值判断SQL语句中敏感字符的来源、转义非可信敏感字符,即可防御SQL注入攻击.基于PHP的Zend引擎实现了系统原型PHPGate,以插件方式实现、易部署.实验证明:PHPGate可精确防御SQL注入攻击,且有效提升污点传播效率,页面应答的时间开销不超过1.6%.

收稿日期:2016-06-16;**修回日期**:2016-08-11
基金项目:国家自然科学基金项目(61572149,61402125)
This work was supported by the National Natural Science Foundation of China (61572149, 61402125).
通信作者:韩心慧(hanxinhui@pku.edu.cn)

关键词 SQL 注入攻击;可信敏感字符;动态污点分析;积极污点分析;编码转换

中图法分类号 TP393.08

SQL 注入(SQL injection)是一种代码注入(code injection)攻击[1-2],其根源是 Web 应用程序中的 SQL 语句通常由程序中可信的常量字符串和用户输入等不可信的外来数据动态拼接而成,而 Web 应用程序中存在输入验证漏洞,导致用户输入等不可信数据未经充分净化、过滤就被数据库引擎当作 SQL 代码片段执行.攻击者可以通过构造精巧的畸形输入,生成恶意 SQL 语句进入数据库执行,以此越权获取数据库中的数据,或通过数据库相关接口直接执行系统指令.

开放式 Web 应用程序安全项目组织(open Web application security project, OWASP)发布的 Web 安全威胁评估报告[3]显示,SQL 注入攻击是当前最严重的 Web 安全威胁.SQL 注入攻击泛滥的原因在于:首先,攻击原理简单、门槛较低,自动化攻击工具众多[4];其次,攻击后得到的数据价值高,例如口令数据库、用户个人隐私等;最后,避免 SQL 注入漏洞需要在 Web 应用程序代码中缜密全面地对用户输入等不可信数据进行验证,或使用参数化查询、存储过程等来约束 SQL 语句语法结构[5-6],但这种安全思维并不普遍存在于程序员群体[7-10].

研究人员基于污点分析来标识 SQL 语句各部分的来源,依赖 SQL 语句语法解析进行 SQL 注入攻击检测.主要分为 3 个步骤:1)污点标记,即对用户输入等不可信数据或可信的常量字符串设置污点标记(前者为消极污点标记 negative taint,后者为积极污点标记 positive taint),其中,标记粒度通常在字符级别,即每个字符对应一个污点状态;2)污点传播(taint propagation),即在程序执行过程中,对带有污点的字符串进行运算、移动操作时将污点标记传播到结果字符串的对应位置;3)污点检查,在 SQL 语句执行点(sink 点)解析 SQL 语句语法结构,分析各部分(如关键词和操作符)的污点信息来检测 SQL 注入攻击.

现有的基于污点分析的 SQL 注入攻击检测方法在可部署性、检测性能、检测精度上均存在不足,难以实际应用:1)现有方案需要修改 Web 应用程序执行引擎,通过在 String 类中增加污点属性域来记录其中每个字符的污点状态,或在内存中专门分配一块空间,以字符地址为索引,以 0,1 为值来记录该字符的污点状态,污点标记及污点传播过程即是对这些属性域或内存空间的赋值与更新,该过程带来不少的额外空间占用和时间性能损耗,并且修改引擎的实现方式不便于实际部署;2)在 Sink 点检查时需对 SQL 语句语法分析[11-12],检测的准确率依赖语法分析的精度[12-13],并且语法分析本身也会带来一定的性能损耗[12].

本文提出了基于敏感字符的 SQL 注入攻击防御方法,无需修改 Web 应用程序执行引擎,无需依赖 SQL 语句语法分析,自动完成 SQL 注入攻击防御.主要贡献包括:

1) 分析 SQL 注入攻击机理,提出了敏感字符和 SQL 安全条件的概念.无需依赖 SQL 语句语法解析进行 SQL 注入攻击检测,仅对来自常量字符串的可信敏感字符进行污点标记和跟踪,在 SQL 语句中识别并转义来自不可信输入的非可信敏感字符,从而保证 SQL 安全条件、防御 SQL 注入攻击.

2) 无需修改 Web 应用程序执行引擎来标记和跟踪污点信息,提出了一种基于编码转换的轻量级污点标记及传播方法.利用编码转换将污点信息直接存储在可信敏感字符的编码值中,节约了污点信息的额外存储空间;污点信息随着字符串操作可直接传播到结果字符串,节约了污点传播的时间开销.该方法广泛适用于 UTF-8 编码的 Web 应用程序.

3) 整个方法实现为一个 PHP 插件——PHP-Gate,自动完成 SQL 注入攻击防御.无需修改 Web 应用程序执行引擎,方便部署.

4) 实验验证防御效果和性能损失.证明 PHP-Gate 可以高效进行污点传播、准确防御 SQL 注入攻击,且页面应答的时间开销不超过 1.6%.

1 SQL 注入攻击机理

Web 应用程序将可信的常量字符串和用户输入等不可信的外来数据动态拼接出 SQL 语句.SQL 注入攻击成功的本质在于控制结构(语法结构)和外来数据混合在同一传输通道中[6],攻击者精心构造的输入没有仅作为查询值,而是破坏了约定的 SQL 语句的语法结构,导致输入数据作为未授权的 SQL 代码片段被执行.如图 1 中,攻击者构造图 1②所示的参数值与图 1①中的常量字符串拼接,从而在图 1③中生成的 SQL 语句中注入了 OR '1'='1' 永真式条件.

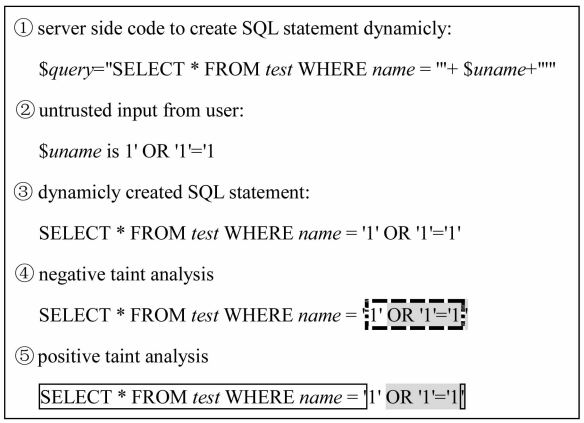


Fig. 1 An example of SQL injection and detection methods based on taint analysis.

图 1 SQL 注入攻击实例与基于污点分析的检测示例

图 1 中④⑤展示了现有的消极或积极污点分析方法的污点检查流程(图 1④中虚线框中为带有消极污点标记的部分,图 1⑤中实线框为带有积极污点标记的部分):1)解析 SQL 语句;2)发现 OR '1'='1' 条件语句含有消极污点标记或者不含积极污点标记,即检测出未授权的 SQL 代码片段(图 1④⑤中阴影部分).

进一步分析图 1 的 SQL 注入攻击实例,SQL 注入攻击分为 2 步:

1) 闭合当前语法结构,通过注入单引号' 来闭合 *name* 查询值;

2) 引入新的语法结构,通过 OR '1'='1 与常量字符串中的单引号' 拼接来构造永真式条件.

在该实例中,攻击者精心注入引号、空格、等号等分隔符,使生成的 SQL 语句的语法和语义发生了变化.

本文的 SQL 注入攻击防御方法基于 2 种现象:

1) SQL 文法是上下文无关文法. 其语法被一些语法相关字符(如空格、分号、引号等)所控制、约束;

2) 数据库引擎最终执行安全的 SQL 语句的语法和语义应与 Web 程序开发时所约定的语法和语义一致. 其充要条件为:最终执行的 SQL 语句中,所有的语法相关字符一一对应于 Web 程序开发时内置的常量字符串中的语法分隔符.

上述条件保证 SQL 语句的语法结构未被攻击者篡改. 由于 SQL 注入攻击并不改变 Web 程序自身的执行逻辑,且不能更改常量字符串中语法相关字符及其顺序,因此上述条件放宽为:SQL 语句中所有的语法相关字符都来自于常量字符串,也可以保证无 SQL 注入攻击发生. 我们称之为"SQL 安全条件".

2 方法设计及关键点

为了便于后文描述,首先进行下述定义:

定义 1. 敏感字符. SQL 文法定义中,对 SQL 语句的语义和语法结构敏感的字符①.

定义 2. 可信敏感字符和非可信敏感字符. 按照字符的不同来源,将敏感字符分为可信敏感字符和非可信敏感字符:可信敏感字符为来自 Web 应用程序常量字符串的敏感字符;非可信敏感字符为来自不可信输入的敏感字符.

基于上述定义,结合 SQL 安全条件,分析如下:

1) 如果 SQL 语句中的敏感字符全部为可信敏感字符,即全部来自可信的常量字符串,则可推出该 SQL 语句中的语法相关字符也一定全部来自常量字符串,符合 SQL 安全条件,不存在 SQL 注入攻击.

2) 如果 SQL 语句含有来自不可信输入的敏感字符,则这些敏感字符可能被数据库引擎解析为语法相关字符,破坏 SQL 语句的语法结构,导致 SQL 注入攻击. 为了保证 SQL 安全条件,避免非可信敏感字符作为 SQL 语句的语法相关字符,可利用数据库引擎特有的转义特性,在这些非可信敏感字符前添加转义符,实现 SQL 注入攻击防御.

3) 为了区分 SQL 语句中敏感字符的来源,对常量字符串中的敏感字符进行积极污点标记(见 2.1 节、2.2 节),使用编码转换的方式将污点信息直接存储在字节值中(见 2.3 节),在程序动态执行过程中进行污点传播,将污点标记传播到生成的 SQL 语句中.

基于敏感字符的 SQL 注入攻击防御方法如图 2 所示:在污点标记时,将标记对象(taint source)限定为常量字符串中的敏感字符(示例中的实线框部分),对其进行编码转换;污点传播过程与程序对可信敏感字符的操作相伴;污点检查过程则在 SQL 语句被送往数据库引擎执行前,将可信敏感字符恢复为标准编码,在非可信敏感字符前添加转义符(本文以反斜线\为例).

① 在 SQL 语句中,敏感字符既可以作为语法相关字符(如图 1 中攻击者输入的空格),也可作为普通字符(如图 1 示例中,若用户输入"张三",其中的空格即为普通字符)

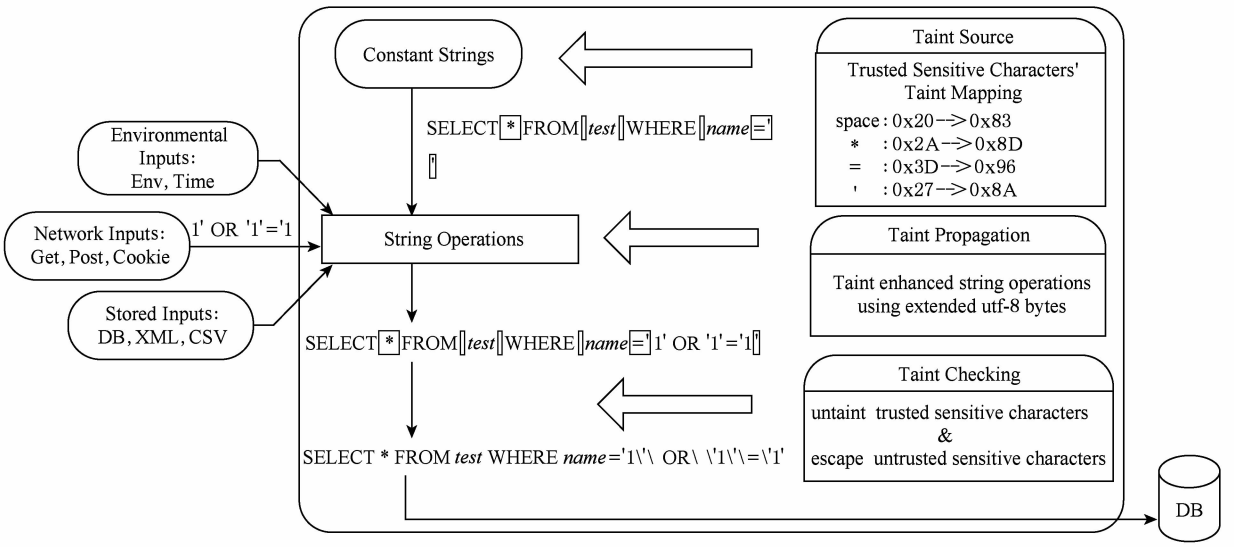


Fig. 2 SQL injection prevention based on sensitive characters.

图 2 基于敏感字符的 SQL 注入攻击防御方法设计

该方法可成功防御 SQL 注入攻击.图 2 的示例中,数据库引擎将字符串中的\和随后字符当作一个转义序列,\和单引号、空格、等号的组合被认作转义序列,数据库引擎将在 *test* 表中查找 *name* 为 1′ OR ′1′=′1 的记录,成功防御 SQL 注入攻击.

该方法不会影响用户与 Web 应用程序、数据库引擎的正常交互.图 2 的示例中,若用户输入“张三”(中间含有非可信敏感字符——空格),经过处理后的查询条件变为 WHERE *name*=′张\ 三′,数据库引擎将字符串中的\和随后空格当作一个转义序列,仍将在 *test* 表中查找 *name* 为“张 三”(中间含有空格)的记录.

2.1 敏感字符集

常见的 SQL 敏感字符有单引号、分号、破折号、反斜杠、@、星号等.若要达到完备的 SQL 注入攻击防御效果,敏感字符集应该包括所有能够改变 SQL 语句的语法和语义的字符.通过对大量 SQL 注入攻击的实例分析,逐一排查 ASCII 字符,可以得到 33 个字符,再另外加上 SQL 语句接受的换行符、回车符、制表符 3 个空白符,构成容量为 36 的敏感字符集合,如表 1 所示.

表 1 的敏感字符是完备的:不包含 SQL 不接受的空白字符、非可见字符以及表 1 中敏感字符的字符串,只能由大小写字母、数字以及中文、韩文等组成,这样的字符串并不能引起 SQL 语句在语法和语义上的变化,攻击者无法通过注入这样的字符串来实现 SQL 注入攻击,进而越权读取数据、越权执行命令.

Table 1 Sensitive Characters and Their Encodings

表 1 敏感字符表,编码转换前/后的字节值

Char	Encoding	Char	Encoding	Char	Encoding
TAB	09/80	)	29/8C	?	3F/98
LF	0A/81	*	2A/8D	@	40/99
CR	0D/82	+	2B/8E	[	5B/9A
space	20/83	,	2C/8F	\	5C/9B
!	21/84	-	2D/90	]	5D/9C
″	22/85	.	2E/91	^	5E/9D
#	23/86	/	2F/92	_	5F/9E
$	24/87	:	3A/93	`	60/9F
%	25/88	;	3B/94	{	7B/A0
&	26/89	<	3C/95	|	7C/A1
′	27/8A	=	3D/96	}	7D/A2
(	28/8B	>	3E/97	~	7E/A3

2.2 积极污点标记可信敏感字符

对敏感字符的污点标记分为 2 种方式:对常量字符串中的可信敏感字符设置积极污点标记和对不可信输入中的非可信敏感字符设置消极污点标记.本文采用积极污点标记方式,其优点在于:实际系统中,不可信输入来自于诸多渠道参数,并没有一个简单、可用的接口来对所有不可信输入进行标记,消极污点标记难以将所有的非可信敏感字符都标记为污点数据;而可信数据以常量字符串形式存在,对 Web 程序常量字符串中的可信敏感字符进行积极污点标记后,所有非可信敏感字符则会以未标记的形式存在,标记结果更加准确.

2.3 编码转换方式记录污点状态

现有的污点标记及传播模型需要 Web 应用程序执行引擎修改 String 类数据结构或者额外分配内存空间来记录污点信息,由于性能损耗较大而难以得到实用.

本文利用 UTF-8 编码特性,通过编码转换来记录污点状态,通过编码值来区分可信敏感字符和非可信敏感字符.该方案减少了污点标记和传播的空间和时间开销:把需要标记的字符的字节值转换为不符合 UTF-8 编码规则的字节值,编码转换之后的字节值自带污点信息,无需额外分配空间记录污点信息;污点传播时,污点信息可被字符串操作自动传播到结果字符串的相应位置;污点检查时,只需扫描字符串,找到非标准的 UTF-8 编码字节即为污点字符.方法属于一种轻量级的污点标记及分析方法,并且适用于所有 UTF-8 编码的 Web 程序.

UTF-8 编码是 Unicode 字符集(U+0000~U+FFFF)的一种编码方式.对于字符 U+0000~U+007F,UTF-8 与基本 ASCII 编码方案一致,采用一个字节表示(二进制表示为 0xxxxxxx);对于之后的 Unicode 字符,UTF-8 采用 2~4 个字节表示,其中首字节由 n 位连续的 1 加 1 位 0 开始(n 为字符编码所需的字节数),其后的每个字节的最高 2 b 固定位为"10".例如 3 B 字符的 UTF-8 编码结构为"1110xxxx 10xxxxxx 10xxxxxx",其中掩码部分即 Unicode 字符的序号.

由于表 1 中的敏感字符在 UTF-8 编码中均是单字节字符,而 UTF-8 单字节字符编码规则中存在形如 1xxxxxxx(0x80-0xFF)的"编码空洞".在标准的 UTF-8 编码字节流中,形如 10xxxxxx 的字节一定是多字节字符的后续字节,而不会是单字节或多字节字符的首字节.根据 UTF-8 编码的上述特性,在单字节字符编码空洞 10xxxxxx 中为可信敏感字符划分出编码空间,使用编码转换的方式进行污点状态记录:即在污点标记阶段,将常量字符串中的可信敏感字符的 UTF-8 编码字节值一一转换为 0x80-0xA3 这些特殊的"非法"编码字节值,具体的编码转换映射关系如表 1 所示.这些非标准 UTF-8 编码的单字节自带污点信息,且可以在程序执行中自动传播到结果字符串中,并能与以 0xxxxxxx 原编码形式存在的、未被污点标记的非可信敏感字符进行区分.

对可信敏感字符的编码转换并没有为 UTF-8 解码带来二义性:本文只用了部分 UTF-8 单字节字符编码"空洞"(0x80-0xBF),即被标记的可信敏感字符具有形式"10xx xxxx".在标准的 UTF-8 编码中,"10"开头的字节不会存在于字符的首字节,而在解码带有污点信息的 UTF-8 字节流时,若一旦遇到首字节就为"10xxxxxx"形式,则该字符一定是经过编码转换后的可信敏感字符.对可信敏感字符的编码转换没有使用编码空洞的另一部分(0xC0-0xFF,具有形式"11xxxxxx",其可能会是多字节字符的首字节),不会引起 UTF-8 解码的二义性.

3 方法描述

第 2 节提出将可信敏感字符的字节值转换为非标准 UTF-8 编码值,这种基于编码转换的污点状态记录方式,广泛适用于采用 UTF-8 编码的 Web 应用程序.

本文的污点标记、污点传播、污点检查过程分别对应着 UTF-8 字节流的编码转换、字节值操作以及编码检查过程.本节详细描述各个过程,为了便于后文描述,首先阐述下述定义.

定义 3. C 为 UTF-8 标准编码的全集,具体分为 4 类:

1) 单字节字符,编码字节为 0xxxxxxx;

2) 2 个字节字符,编码字节为 110xxxxx 10xxxxxx;

3) 3 个字节字符,编码字节为 1110xxxx 10xxxxxx 10xxxxxx;

4) 4 个字节字符,编码字节为 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx.

定义 4. SC 为所有敏感字符的 UTF-8 编码集合.SC 中共 36 个元素,均为单字节编码,具体如表 1 所示(形如 0xxxxxxx).

定义 5. SSC 为非标准的单字节字符编码,共 36 个元素,值为 10000000~10100011 (均具有形式 10xxxxxx)

定义 6. 编码转换 map 将 SC 中的标准编码一一映射到 SSC 中的非标准编码,具体的映射关系如表 1 所示,定义如下:

$$map: SC \rightarrow SSC,$$

$$\varepsilon \in SC \Rightarrow map(\varepsilon) \in SSC,$$

$$\varepsilon \in SC \wedge \gamma \in SC \wedge \gamma \neq \varepsilon \Rightarrow map(\gamma) \neq map(\varepsilon).$$

定义 7. 编码恢复 $remap$ 将 SSC 中的非标准编码还原为 SC 中的标准编码,具体的映射关系如表 1 所示,定义如下:

$$remap: SSC \rightarrow SC,$$
$$\varepsilon \in SSC \Rightarrow remap(\varepsilon) \in SC,$$
$$\varepsilon \in SSC \wedge \gamma \in SSC \wedge \gamma \neq \varepsilon \Rightarrow remap(\gamma) \neq remap(\varepsilon).$$

上述定义具有 4 个性质:

性质 1. $SC \subseteq C$.

性质 2. 不可伪造性. $SSC \cap C = \varnothing$ 使非标准编码与标准 UTF-8 编码完全区分,即非标准编码不能被 UTF-8 标准编码伪造.

性质 3. 编码转换 map 具有编码唯一性,即:

$$\forall \varepsilon \in SC \wedge \forall \gamma \in SC \wedge \gamma \neq \varepsilon \Rightarrow map(\gamma) \neq map(\varepsilon).$$

性质 4. 编码恢复 $remap$ 具有解码唯一性,即:

$$\forall \varepsilon \in SSC \wedge \forall \gamma \in SSC \wedge \gamma \neq \varepsilon \Rightarrow remap(\gamma) \neq remap(\varepsilon).$$

3.1 基于编码转换的污点标记

算法 1 完成对可信敏感字符的积极污点标记.

算法 1. 污点标记算法 TaintSC.

输入:UTF-8 常量字符串的集合 *ConststringSet*;

输出:编码转换后的常量字符串集合.

① for each Q in *ConststringSet*:
② character_list $cl = utf8\_standard\_decode(Q)$;
③ for each μ in cl:
④ $Mark(\mu)$;
⑤ end for
⑥ end for

步骤①遍历 Web 应用程序中常量字符串的集合 *ConststringSet*,选定一个 UTF-8 常量字符串,其字节流为 Q.

步骤②对 Q 进行 UTF-8 标准解码. 即按照 C 的编码集合,对 Q 进行划分,划分出 1 个字节组序列 cl,其中每个字节组含有 1~4 个字节,对应着 1 个 UTF-8 字符.

步骤③遍历步骤②中字节组序列中的每个 UTF-8 字符.

步骤④对每个 UTF-8 字符,用定义 8 中的函数 $Mark$ 将敏感字符的标准编码字节值转换为非标准编码字节值,其他字符的编码字节值则保持原样.

定义 8. 函数 $Mark$:

$$Mark: C \rightarrow SSC \cup \{C - SC\},$$
$$Mark(\mu) = \begin{cases} map(\mu), \mu \in SC \\ \mu, \mu \in \{C - SC\} \end{cases}. \quad (1)$$

污点标记算法 TaintSC 完成了对常量字符串中可信敏感字符的标记,可信敏感字符的编码字节值均被 SSC 中的非标准编码字节值替换,常量字符串中其余字符则保留原编码字节值,被标记后的常量字符串的编码空间为 $SSC \cup \{C - SC\}$.

3.2 基于编码转换的污点传播

Web 应用程序常量字符串中的可信敏感字符经过 TaintSC 算法被转换为 SSC 中的非标准编码字节值,而用户输入等不可信数据中的非可信敏感字符则以 SC 中的标准编码形式存在. 程序执行时,将二者混合在一起操作,字符串的编码空间为 $SSC \cup C$.

污点传播要求在字符串操作中跟踪、存储和更新结果字符串中相应的污点信息. 可信敏感字符以编码转换的方式在字节值中自动携带污点信息,污点传播过程与程序对可信敏感字符的操作相伴. 字符字节值中自动携带的污点信息会随着程序执行自动传播,不需要额外的跟踪算法. 但是,由于污点标记后的字节流不符合标准 UTF-8 编码规则,程序中的字符串函数需分为 3 种情况进行处理:

1) 拼接、分割类函数

该类函数(如 join,substr 等)对字符串参数做拼接、分割处理. 编码转换的污点标记方式并不改变字符串长度和字符位置,因此无需对这些函数进行额外处理,参数中可信敏感字符的非标准编码字节可随着函数执行自动赋值到结果字节流中相应位置.

2) 数值操作类函数

该类函数如 md5 等,直接对整个字符串取值后进行字面值操作. 对于这类函数,在函数执行前需对参数中的可信敏感字符进行编码恢复(即污点净化),从而保证得到正确的函数执行效果.

3) 含有字符匹配逻辑的函数

该类函数分为 2 个步骤:首先通过字符(串)匹配逻辑确定操作位置;然后在此位置上对字节流进行增(如 addcslashes 函数增加反斜线)、删(如 trim 函数删除指定字符)、改(如 replace 函数用指定字符串替换)等操作.

字符匹配过程中,由于编码转换的污点标记方式使每个敏感字符根据是否来自常量字符串存在 2 种字节值,所以需对该逻辑进行修改:若 2 个字符 α, β(无先后顺序)满足式(2)所述条件,则认为二者匹配:

$$(\alpha, \beta \in C \wedge \alpha = \beta) \vee (\alpha, \beta \in SSC \wedge \alpha = \beta) \vee (\alpha \in SC \wedge \beta \in SSC \wedge \alpha = remap(\beta)). \quad (2)$$

而增、删、改操作会自动将参数中的污点标记赋值到结果字节流中相应位置,实现污点的自动传播,所以无需做额外处理.

3.3 基于编码转换的污点检查

在 SQL 语句传入数据库引擎执行前,分析 SQL 语句的 UTF-8 字节流,进行污点检查,检查 SQL 语句中所有的敏感字符是否都带有污点标记(*SSC* 中的编码形式,即可信敏感字符)以及是否存在未带有污点标记的敏感字符(*SC* 中的原编码形式,即非可信敏感字符),算法如下:

算法 2. 污点检查算法 SQLCheck.

输入:动态拼接出的 SQL 语句,其字节流为 Q;

输出:编码恢复后的 SQL 语句字节流,非可信敏感字符的下标集合 $UntrustIndex$.

① $UntrustIndex:=\varnothing$;
② character_list $cl=utf8\_extended\_decode(Q)$;
③ for each μ in cl / * $\mu\in\{SSC\cup C\}$ * /
④ if $\mu\in SSC$
⑤ $remap(\mu)$;
⑥ end if
⑦ if $\mu\in SC$
⑧ $UntrustIndex.add(\mu.index)$;
⑨ end if
⑩ end for

步骤②对 Q 进行 UTF-8 扩展解码. 即按照 $SSC\cup C$ 中的编码集合对 Q 进行划分,划分出 1 个字节组序列 cl. 其中每个字节组含有 1～4 个字节,对应着一个 UTF-8 字符或带污点标记的可信敏感字符.

步骤③遍历步骤②中字节组序列中的每个字符 μ.

步骤④～⑨对步骤③的每个 μ,将可信敏感字符($\mu\in SSC$)的非标准编码字节值恢复为标准编码字节值,非可信敏感字符的编码字节值($\mu\in SC$)则保持原样,将其下标加入 $UntrustIndex$ 中,对于 $\mu\in\{C-SC\}$的字符,字节值保持原样.

3.4 SQL 注入攻击自动防御

经过 SQLCheck 算法的处理,SQL 语句中的可信敏感字符已经恢复为标准的 UTF-8 编码字节值,整个 SQL 语句为标准的 UTF-8 编码. 通过算法 3 完成 SQL 注入攻击的自动防御.

算法 3. SQL 注入攻击自动防御算法 SQLInjectionPrevention.

输入:SQLCheck 处理后的 SQL 语句字节流 Q 以及非可信敏感字符的下标集合 $UntrustIndex$;

输出:在非可信敏感字符前添加转义符,执行 SQL 语句.

① if $UntrustIndex=\varnothing$;
② SQL_exec(Q);
③ else
④ for each i in $UntrustIndex$
⑤ prepend '\' before $Q[i]$;
⑥ end for
⑦ SQL_exec(Q);
⑧ end if

步骤①～②若发现 $UntrustIndex=\varnothing$,则说明 SQL 语句中不含有非可信敏感字符,SQL 语句中的敏感字符全部来自可信的字符串常量,该 SQL 语句符合 SQL 安全条件,不存在 SQL 注入攻击,被直接送往数据库执行.

步骤③～⑦若 $UntrustIndex\neq\varnothing$,则 SQL 语句含有来自不可信输入的敏感字符,这些非可信敏感字符可能破坏 SQL 语句语法结构,造成 SQL 注入攻击. 在这些非可信敏感字符前添加转义符\来自动防御 SQL 注入攻击(由于添加转义符会使 SQL 语句长度发生变化,步骤④～⑥仅表示抽象逻辑).

4 PHPGate 系统实现

针对 PHP 语言进行原型系统实现. 本文的方法无需修改 PHP 执行引擎源码,而是将上节所述方法实现为 PHP 执行引擎的一个插件——PHPGate. 插件的实现方式较易部署,同时 PHP 执行引擎是 C 语言编写,插件也是 C 语言编写,执行效率高.

在一个典型的 Web 服务器中,PHP 程序被执行引擎——Zend 解释执行. 如图 3 实线方框所示,过程分为 2 步:首先,生成中间码(byte codes),其中,PHP 程序中的常量字符串被解释为中间码的参数;然后,执行中间码,执行时调用相应的处理函数(handler)完成功能.

Zend 引擎提供了完整的插件开发 API,使开发者可以在 PHP 解释、执行过程中插装代码,监控并操作应用层代码的执行细节. 如图 3 虚线方框所示,PHPGate 通过遍历中间码及其参数实现污点标记,通过对 Zend 引擎中一系列函数的 hook 实现污点传播、污点检查.

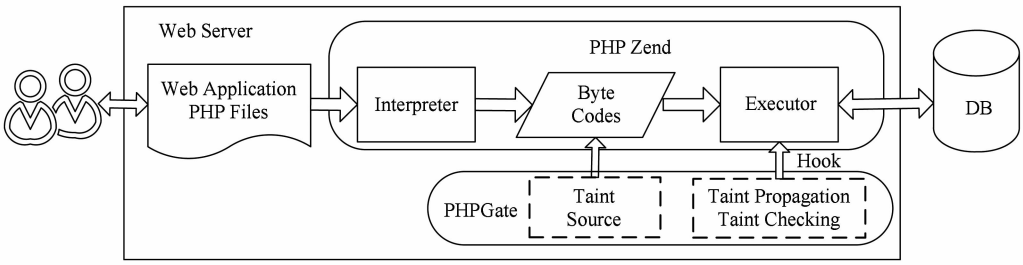


Fig. 3 PHPGate implementation and working process.

图 3 PHPGate 系统工作流程

4.1 可信敏感字符标记

可信敏感字符的污点标记包括 2 个步骤:常量字符串提取和可信敏感字符编码转换. 常量字符串提取负责从中间码的参数中找到常量字符串. 可信敏感字符编码转化则将常量字符串中敏感字符的标准编码字节值转变为表 1 中的非标准编码字节值.

PHPGate 通过 Zend 引擎中 zend_compile_file 函数实现常量字符串提取和可信敏感字符编码转换. Zend 引擎通过 zend_compile_file 函数把 PHP 程序解释成一系列中间码,PHPGate 通过插件的相应接口 hook 该函数,在中间码生成后遍历所有的中间码,检查每一条中间码的参数:如果参数类型为字符串且具有属性 IS_CONST,则该参数为 PHP 程序中的常量字符串,按照 3.1 节所述的 TaintSC 算法完成可信敏感字符编码转换.

4.2 污点传播及净化的实现

1) 内置字符串函数

PHPGate 对 PHP 内置字符串函数的 handler 进行 hook,按照 3.2 节所述的 3 种不同分类分别实现无操作、参数污点净化(即编码恢复)以及重定义字符匹配逻辑.

2) 特殊的中间码

可信敏感字符的编码转换改变了一些中间码参数的字节值,为了保证程序正常逻辑不被破坏,还需要对下述 5 个中间码进行参数净化处理:

ZEND_INCLUDE_OR_EVAL, ZEND_ECHO, ZEND_PRINT, ZEND_DO_FCALL_BY_NAME, ZEND_DO_FCALL.

前 3 条中间码在执行时无法处理可信敏感字符的非标准编码形态,PHPGate 修改它们的 handler,使之指向一个 wrap 函数. wrap 函数首先对中间码的字符串参数进行污点净化,将可信敏感字符编码恢复;接着调用原来的 handler 完成操作;最后在 handler 执行完后将参数中的可信敏感字符重新编码转换为污点形态(污点重标记),保证污点的正确传播.

后 2 个中间码的语义为根据参数指明的函数名查找对应的内置函数. 如果这 2 个中间码的字符串参数(即函数名)中包含非标准编码,也会引起逻辑错误,导致找不到所调用的函数,所以也需对参数进行污点净化及污点重标记.

4.3 污点检查及 SQL 注入攻击防御的实现

在 SQL 语句送入数据库引擎前,需要对所有的敏感字符进行检查:对于污点形态的敏感字符,需要编码恢复;对于正常形态的敏感字符,需要进行相应处理,如在其之前加入'\'字符使其失去改变 SQL 语义的功能.

与 hook 内置字符串函数的方式类似,PHPGate 对数据库操作函数指定了一个新的 handler,实现 SQLCheck 算法和 SQLInjectionPrevention 算法. 最后将处理过的 SQL 语句传给数据库引擎执行.

4.4 伪标记问题的处理

4.1 节至 4.3 节实现的 SQL 注入攻击防御方案基于一个前提:程序接受的用户输入都是 UTF-8 标准编码的字符串,即用户无法输入非标准 UTF-8 单字节. 在 PHP 运行过程中会遇到类似 md5 等的 Hash 函数,这些 Hash 函数会产生任意字节值. 这些字节可能会被解释为可信敏感字符,为了避免对这样的字节进行污点净化,PHPGate 在所有的净化过程中对字符串加入 UTF-8 扩展解码合法性检查. 如果字符串不是一个合法的扩展 UTF-8 编码字符串,则不对这个字符串进行污点净化操作.

4.5 性能优化

4.1 节至 4.4 节实现的 PHPGate 可以保证逻辑正确运行,但是性能有待优化.

在类似 ZEND_ECHO 的参数污点净化和污点重标记处理中,如果反复申请内存以保存字符串的污点标记形式又在之后释放,将引起很大的性能开销.

PHPGate 使用一种基于栈的方法来优化. PHPGate 维护一个额外的“标记栈”来辅助污点净化、污点重标记. 标记栈的运行和参数污点净化、污点重标记过程同步. 每当 PHPGate 进行污点净化操作时,PHPGate 向标记栈压入额外信息,额外信息包括每个参数的:

1) 各个污点字符下标;

2) 字符串指针;

3) 污点字符数.

最后,将参数数目压栈.

如“<?php echo "hello'w'orld"; ?>”,2 个单引号在污点标记过程中被编码转换为 0x8A;在执行 ECHO 时,PHPGate 会先对参数进行污点净化,同时在标记栈依次压入 5 和 7(参数字符串中污点字符下标)、字符串参数指针、2(污点字符数为 2)、1(参数数目为 1). 在 ECHO 操作执行后,PHPGate 依据标记栈上弹出的信息直接对参数字符串中的可信敏感字符进行污点重标记,避免了多次申请内存来分别保存污点净化和污点重标记前后的字符串参数.

5 实验与分析

为了测试 PHPGate 对 SQL 注入攻击的实际防御效果以及带来的性能损失,搭建了 LAMP(Linux,Apache,MySQL,PHP)框架,其中:操作系统版本为 Debian Linux 7,Linux 内核为 3.2.0-4-amd64,Apache 版本为 Apache 2.2.24,MySQL 版本为 5.5.35,PHP 版本为 5.4.21.

防御效果测试中,一方面使用一个已有的 SQL 注入攻击测试集 T;另一方面重现了针对知名 PHP 应用程序的 SQL 注入攻击,观察开启 PHPGate 之后攻击是否成功.

性能测试中,首先采用 Zend 自带的 2 个测试集:ZendBenchmark(bench.php)和 PHP Benchmark Script 测试 PHPGate 在不同类型操作中的性能损失;其次,测试 SQL 注入攻击测试集 T 的页面应答时间开销;最后,针对真实页面测试 PHPGate 的页面应答时间开销.

5.1 防御效果实验及讨论

1) 防御测试集 T 中的 SQL 注入攻击

选取了文献[8]和文献[12]的 SQL 注入攻击测试集 T,如表 2 所示,包含 3 个 PHP 应用、19 个 SQL 注入漏洞及攻击. 实验表明,PHPGate 能有效防御这 19 个 SQL 注入攻击.

Table 2 Effectiveness of SQL Injection Prevention on Benchmark *T*

表 2 测试集 *T* 中的 SQL 注入攻击防御有效性

Web Application	# of Attacks	# of Prevented
schoolmate	6	6
webchess	12	12
faqforge	1	1

表 3 选取了其中的 2 个 SQL 注入攻击实例,描述 PHPGate 对 SQL 注入攻击的防御效果(下划线为用户输入). 示例 1 中,MySQL 数据库引擎将反斜线和其后字符当作转义序列,查找值为 junk" or 1=1--的记录,从而成功防御了 SQL 注入攻击. 示例 2 中,反斜线未在字符串中,MySQL 数据库引擎将反斜线和其后字符当作 2 个独立字符,SQL 语句语法解析失败后无法执行,从而阻止恶意 SQL 语句的执行,防御 SQL 注入攻击.

Table 3 Demonstration of SQL Injection Prevention

表 3 SQL 注入攻击的防御效果示例

Demo	SQL Statements without PHPGate	SQL Statements with PHPGate
demo1	SELECT *password* FROM *users* WHERE *username*="junk" or 1=1 -- "	SELECT *password* FROM *users* WHERE *username*="junk\"\ or\ 1\=1\ \-\-\ "
demo2	SELECT *aperc*, *bperc*, *cperc*, *dperc*, *coursename* FROM *courses* WHERE *courseid*=5 or 1=1	SELECT *aperc*, *bperc*, *cperc*, *dperc*, *coursename* FROM *courses* WHERE *courseid*=5\ or\ 1\=1

2) 防御真实 SQL 注入攻击案例

从 ExploitDB[14] 选择了 5 个公开的针对 PHP 应用的 SQL 注入攻击样本,安装相应版本的 PHP 应用,搭建实验环境并且模拟攻击.

表 4 列举了这 5 个攻击样本所针对的漏洞编号和应用名称. 实验表明:PHPGate 能有效防御这 5 个 SQL 注入攻击.

3) 防御能力对比

PHPGate 不依赖 SQL 语句的语法分析进行攻击检测,避免了由于语法分析策略精度不够造成的

漏报.对于图 4 中的 SQL 注入攻击样例(下划线为用户输入),文献[13]中的语法分析策略不能准确检测出 SQL 注入攻击,而 PHPGate 通过转义 SQL 语句的非可信敏感字符,成功防御 SQL 注入攻击.

Table 4 PHPGate Prevent SQL Injection Attacks in the Wild

表 4 PHPGate 防御真实 SQL 注入攻击

CVE/OSVDB	Web Application	Prevented
OSVDB-103365	osCommerce 2.3.3.4	√
CVE-2013-6936	MyBB Ajaxfs 2 Plugin	√
OSVDB-103126	Joomla 3.2.1	√
CVE-2007-4653	PHPBB 2	√
CVE-2003-1244	PHPBB 2	√

```
SELECT name FROM account WHERE id=exit()
```

Fig. 4 A case of SQL injection.

图 4 一个 SQL 注入攻击样例

4) PHPGate 局限性

下划线_和百分号%是 SQL 语句的通配符,MySQL 数据库引擎仅在 Like 匹配语句中才将字符串中的_和\%当作转义序列.因此,PHPGate 在处理非可信的_和%时,需要使用辅助手段(如 SQL 语句语法分析)来判断它们所在的语法结构并予以相应处理.

5.2 PHPGate 性能测试

1) Zend 测试集的性能测试

图 5 展示了针对 ZendBenchmark 的性能测试结果.横坐标为 18 个测试例,柱状图(左坐标轴,单位为秒)显示了在关闭和开启 PHPGate 的 2 种情况下运行时间的差别,深色柱为不加载 PHPGate 时的运行时间,浅色柱为开启 PHPGate 的运行时间,折线图显示了开启 PHPGate 后性能损失随不同测试例的变化曲线(右坐标轴).可以看出,在 simplecall,strcat(200000)和 simpleucall 这 3 个测试例上 PHPGate 的损失较大,分别为 5.84%,10.91%和 11.21%,而在其他测试例上的性能损失均小于 2%.此外,有些结果的性能损失为负数并不能说明开启 PHPGate 后运行速度比关闭时更快,而是说明 PHPGate 在这些操作上的性能影响小于系统影响、外界环境影响.

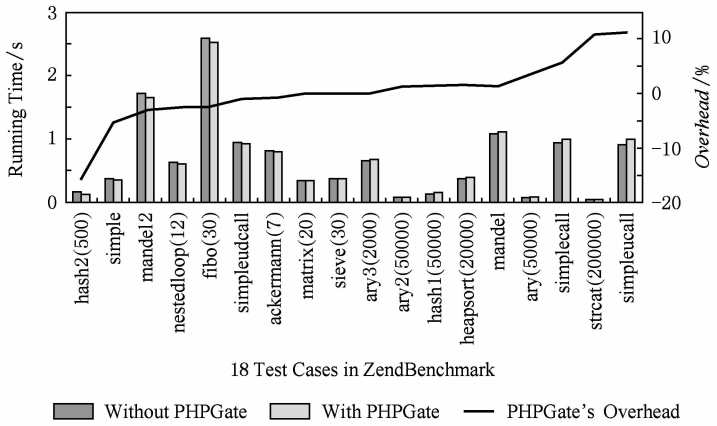


Fig. 5 Overhead on ZendBenchmark.

图 5 ZendBenchmark 性能损失

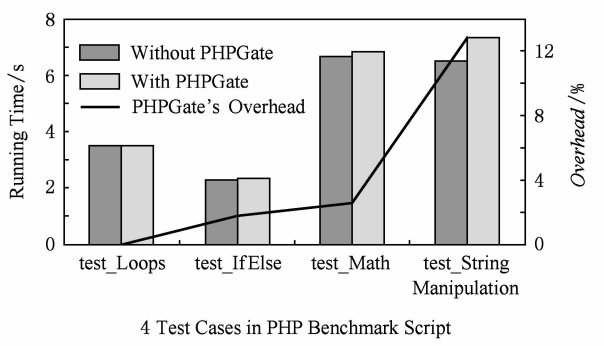


Fig. 6 Overhead on PHP Benchmark Script.

图 6 PHP Benchmark Script 性能损失

图 6 的 PHP Benchmark Script 从数学运算、字符串操作、循环和分支 4 个方面对 PHPGate 的性能进行测试.图 6 展示了针对 PHP Benchmark Script 的测试结果.可以看出,字符串操作测试例上 PHPGate 的损失较大,但仍在可接受的范围内(12.76%),在其他测试例上的性能损失均很小.

表 5 对比了 PHPGate 与 WASP[11,15]在同类型字符串操作函数的性能损失.WASP 对 Java 中 String 类重新进行封装,在新增的属性域中存储每个字符的污点信息,污点传播需要对参数污点属性域逐一遍历,并按照污点传播策略对结果字符串的

污点属性域逐一赋值.该过程对性能损失影响较大,实验数据表明,WASP 在字符串相关操作上的性能损失从 125%～7100%不等.而本文的基于编码转换的污点状态记录方式,污点信息直接存储在字节值中,随着字符串操作直接赋值到结果字符串中.对比看出,本文的污点标记及传播方法有效降低了字符串操作函数上的性能损失.

Table 5 Overhead Comparison of Two Methods on String Manipulation

表 5 字符串操作性能比较

Method	String Functions	*Overhead*/%
WASP[11,15]	String.Concat(String)	170.83
	StinrgBuilder.Append(StringBuilder)	7 100.00
	StinrgBuilder.Append(String)	2 500.00
PHPGate	Stringcat	10.91
	concat(.)	12.78

2) 测试集 T 上的性能损失及比较

选取了文献[12]的 SQL 注入攻击测试集 T,使用脚本在服务器本机对 Web 应用的首页发送访问请求,然后分别统计加载和不加载 PHPGate 的应答时间.由于在服务器本机进行测试,网络延迟可以忽略不计,而且网络延迟在 PHPGate 关闭和开启的情况下可以认为一致,不影响测试结果.对每个页面进行了 50 次请求,然后统计应答时间的平均值以及时间损耗.

表 6 列出了测试集 T 中 Web 应用的代码规模(line of code, LOC)以及使用 PHPGate 带来的性能损失,对比列出了文献[12]的性能损失.

Table 6 Overhead Comparison on Benchmark *T*

表 6 测试集 *T* 中的页面应答时间性能比较

Application	LOC	*Overhead*/%		
		diglossia[12] without Parser	diglossia[12] with Parser	PHPGate
schoolmate	7 024	0.4	12.5	≈0
webchess	5 780	3.1	8.5	≈0
faqforge	1 520	6.2	13.5	≈0

diglossia[12]使用污点分析结合 SQL 语句语法分析进行 SQL 注入攻击检测.表 6 中的列 3 和列 4 显示了 diglossia[12]在污点标记、传播中带来的性能损耗以及加上 SQL 语句语法解析带来的性能损耗.可以看出:一方面,diglossia[12]的污点分析在测试集 T 上带来了一定的性能损耗;另一方面,SQL 语句语法分析影响了 diglossia[12]在测试集 T 上的性能.

在测试集 T 上,PHPGate 的性能损失几乎为 0:一方面在于测试集 T 中的页面相对简单,字符串操作较少,并且 PHPGate 是轻量级的污点标记及传播方案(如表 5 所示);另一方面,PHPGate 无需依赖 SQL 语句语法解析,直接转义非可信敏感字符进行 SQL 注入攻击防御.

3) 真实页面的性能损失

在使用测试集进行性能测试之外,还针对真实页面对 PHPGate 进行性能评估.使用 Python 脚本在服务器本机对测试页面以 POST 方式发送登陆请求,然后统计应答时间.对每个页面进行了 10 000 次请求,然后统计应答时间的平均值.表 7 展示了使用 PHPGate 前后页面访问服务器应答时间的测试结果.可以看出,页面访问的应答时间增加比例均在 1.6%之内.

Table 7 Overhead on Popular Web Applications

表 7 真实页面的应答时间性能损耗

Page	Application	Response Time/s		*Overhead*/%
		Original	with PHPGate	
ucp.php	phpBB3	0.258	0.261	1.16
login.php	phpBB2	0.217	0.22	1.38
member.php	Mybb	0.308	0.311	0.97
login.php	oscommerce	0.189	0.192	1.59
Average				1.28

6 相关工作

围绕 SQL 注入攻击防御的研究一直在持续进行[16-18].研究人员尝试根据一定的策略来推理出 SQL 语句中的可信和不可信部分[19-22],这种方式依赖于推理策略的完备性,准确性不能得到保证.而基于污点分析的方案能通过污点标记、污点传播直接、准确地区分 SQL 语句中的可信和不可信部分,相对比较可靠,主要分为静态污点分析和动态污点分析 2 种.

静态污点分析中,Jovanovic 等人提出的 Pixy[23]采用流敏感、过程间和上下文敏感的静态数据流分析方法来挖掘 Web 应用漏洞.Livshits 和 Lam[24]利用静态数据流分析技术检查经过标记的用户输入是否到达 SQL 语句执行点,以此来挖掘 SQL 注入漏洞.Wassermann 和 Su 将 SQL 语句构造过程抽象为一个上下文无关文法,用静态污点分析方法判断该文法中的非终止符是否与用户输入相关[25].

Dahse 和 Holz 采用静态污点分析方法挖掘二阶式注入漏洞[26],更进一步利用对 900 多个 PHP 内置函数的分析和模拟来进行更精确的静态污点分析[27].但由于静态污点分析方法无法很好地处理 Web 应用程序语言的动态特性,所以会引入较高的漏报,加上其需要保守式的程序分析,所以也会带来较多的误报.

动态污点分析方法则能更好地应对程序的动态特性:Nguyen-Tuong 等人[28]修改 PHP 引擎中 String 类的数据结构和函数操作,在字符级别对不可信数据进行污点标记与传播,在 Sink 检查点分析 SQL 语句的语法结构,不允许 SQL 敏感字符和关键字来自不可信数据. Pietraszek 和 Berghe 的 CSSE[29]则是在 PHP 引擎中使用一个以变量指针为索引、以 bitmap 为值类型的查询表来实现对不可信数据逐字符的污点标记. Xu 等人[30]基于 C 语言代码的自动转换,动态维护了一个以单个字节的地址为索引、以 0 或 1 为值的查询表,通用地对 C 语言程序增强了污点标记与跟踪功能,并对 PHP 引擎进行了实现.

上述方法属于消极污点分析,即对不可信输入进行污点标记,并在程序执行过程中对这些污点标记进行传播.但是由于不可信输入的类型和来源均比较广泛[29],对这些数据的漏标记很容易造成漏报.而相对来说,可信数据的来源一般比较固定,如程序员写在 Web 程序中的常量字符串[29,31].基于这一思想,研究人员提出了积极污点分析的方法来检测 SQL 注入攻击:Halfond 等人[11,15]提出了 WASP,对程序中常量字符串设置可信标记并动态跟踪该可信标记的传播,通过该标记来区分生成的 SQL 语句中的可信部分和不可信部分.由于所有的不可信输入均以未标记形式存在,该类方法能够避免漏报,较精确地检测 SQL 注入攻击.

上述积极/消极的动态污点分析方法,均通过修改数据结构或者分配内存的方式来记录污点信息,性能损耗较大,如 Nguyen-Tuong 等人[28]在字符串相关函数上的性能损失达到 40%~76%,而 WASP[11,15]在字符串相关函数上损失也很大,其中 StringBuilder.Append 函数的运行时间增加了 71 倍.

近年来,研究人员提出基于边界标记、字符随机化、字符/编码转换的方式来标识可信数据/非可信数据,实现轻量级的污点分析:

在边界标记方案中,Su 和 Wassermann 在 SQLCHECK 中[13]用程序随机选的 4 个字符起始标记和结束标记来对不可信的用户输入进行标识,将 SQL 语句的上下文无关文法转换为一个支持该起始标记和结束标记的上下文无关文法,若在检查点不能根据新的文法对带有起始和结束标记的 SQL 字符串进行解析,则说明用户输入影响了 SQL 的语法结构,从而判定发生了 SQL 注入攻击.李舟军等人的 PHPHard[32]则基于类似的边界标记思想,积极污点标记 PHP 应用中常量字符串和内联 HTML 代码,通过边界标记的传播来动态跟踪响应页面的安全区间,用来检测跨站脚本攻击.

在随机化方案中,Boyd 和 Keromytis 认为影响 SQL 语句语义结构的关键词必须来自于程序员写的常量字符串,而不能来自不安全的用户输入,他们设计的 SQLrand[33]是一种类似指令集随机化的防护方案,将常量字符串中的 SELECT、WHERE 等关键词后附加一个随机整数进行随机化,通过随机化来标记可信的关键词,然后在应用程序和数据库之间设置一个中间代理,该代理基于一个支持随机化关键词的 SQL 语法解析器来检查 SQL 语句的合法性并将其去随机化后传输给数据库.攻击者可以猜测作为污点标记的随机整数并尝试构造攻击,此外,由于需要一个代理在应用程序和数据库之间进行通信,SQLrand 的部署也受到限制.

在字符/编码转换方案中,Mui 等人[31,34]采用 ASCII 编码中空置的最高位来标识污点信息,对不可信的用户输入进行标记,并修改 SQL 语法解析器,检查敏感节点是否来自不可信输入.该污点标记方式适用于 ASCII 单字节编码的字符串,但在使用 UTF-8 编码的 Web 应用程序中会引起二义性. Son 等人提出的 Diglossia[12]将 Web 程序常量字符串中的字符逐一映射到非英文字符(如韩文字符),通过影子字符映射进行污点标记,但攻击者可以猜测该映射关系并尝试构造能绕过检测的攻击. Ahuja 等人[35]和 Heart 等人[36]则将不可信字符直接替换成其二进制或十六进制表示(如将 A 替换成 41),该种方式容易与正常的数字串混淆,从而引起程序功能异常. Zekan 等人[37]和 Masri 等人[38]则针对 unicode 编码的字符串,利用一个固定的 OFFSET,将被标记的字符映射到 unicode 编码的闲置空间.本文提出的基于编码转换的污点标记方案,适用 UTF-8 编码的 Web 应用程序,将污点信息直接存储在 *SSC* 中的非标准单字节编码中,标记后的字符串长度未

受影响,攻击者无法向 Web 程序中输入 SSC 中的非标准单字节编码,SSC 只能来源于被标记的可信敏感字符,且 $SSC \cap C = \varnothing$,防御方案的安全性和可靠性得到了保证.

7 结 论

本文提出了基于敏感字符的 SQL 注入攻击防御方法,结果表明该方法能够高效、精确防御 SQL 注入攻击.该方法首先基于积极污点分析思想,将污点标记范围从可信的常量字符串缩小到可信的 SQL 敏感字符;其次利用 UTF-8 单字节字符的编码空洞,采用编码转换的方式对可信敏感字符进行污点标记,避免了额外开辟空间存储污点信息;最后利用污点信息区分可信敏感字符和非可信敏感字符,在非可信敏感字符前添加转义符,防御 SQL 注入攻击.该方法实现为 PHP 插件,易于部署,可以自动和精确防御 SQL 注入攻击,广泛适用于 UTF-8 编码的 Web 应用程序.

本文的防护效果实验验证了 PHPGate 对 SQL 注入攻击防御的有效性.尽管实验中的攻击案例都是利用已知漏洞来构造攻击输入,但该方法同样可以对利用未知漏洞的 SQL 注入攻击进行防御,因为无论漏洞成因以及攻击输入如何构造,污点信息均以非标准单字节编码的形式传播到 SQL 语句,可以通过污点信息区分可信和非可信敏感字符,进而通过对非可信敏感字符的转义来防御针对未知漏洞的 SQL 注入攻击.

参 考 文 献

[1] Ray D, Ligatti J. Defining code-injection attacks [C] //Proc of the 39th Annual ACM SIGPLAN-SIGACT Symp on Principles of Programming Languages. New York: ACM, 2012: 179-190

[2] Ray D, Ligatti J. Defining injection attacks [G] //LNCS 8783: Proc of the 17th Int Conf on Information Security. Berlin: Springer, 2014: 425-441

[3] The Open Web Application Security Project. OWASP TOP10 critical Web application security risks [EB/OL]. [2015-11-05]. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[4] Zhao Yufei, Xiong Gang, He Longtao, et al. Approach to detecting SQL injection behaviors in network environment [J]. Journal of Communications, 2016, 37(2): 88-97 (in Chinese)
(赵宇飞, 熊刚, 贺龙涛, 等. 面向网络环境的 SQL 注入行为检测方法[J]. 通信学报, 2016, 37(2): 88-97)

[5] The Open Web Application Security Project. SQL Injection Prevention Cheat Sheet [EB/OL]. [2015-11-05]. https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet

[6] Clarke J. SQL Injection Attacks and Defense [M]. Amsterdam: Elsevier, 2012
(Clarke J. SQL 注入攻击与防御[M]. 施宏斌, 叶愫, 译. 2 版. 北京: 清华大学出版社, 2013)

[7] Balzarotti D, Cova M, Felmetsger V, et al. Saner: Composing static and dynamic analysis to validate sanitization in Web applications [C] //Proc of the 2008 IEEE Symp on Security and Privacy. Piscataway, NJ: IEEE, 2008: 387-401

[8] Kieyzun A, Guo P J, Jayaraman K, et al. Automatic creation of SQL injection and cross-site scripting attacks [C] //Proc of the 31st Int Conf on Software Engineering. Piscataway, NJ: IEEE, 2009: 199-209

[9] Martin M, Lam M S. Automatic generation of XSS and SQL injection attacks with goal-directed model checking [C] //Proc of the 17th USENIX Security Symp. Berkeley: USENIX Association, 2008: 31-43

[10] Appelt D, Nguyen C D, Briand L C, et al. Automated testing for SQL injection vulnerabilities: An input mutation approach [C] //Proc of the 2014 Int Symp on Software Testing and Analysis. New York: ACM, 2014: 259-269

[11] Halfond W G J, Orso A, Manolios P. Using positive tainting and syntax-aware evaluation to counter SQL injection attacks [C] //Proc of the 14th ACM SIGSOFT Int Symp on Foundations of Software Engineering. New York: ACM, 2006: 175-185

[12] Son S, Mckinley K S, Shmatikov V. Diglossia: Detecting code injection attacks with precision and efficiency [C] //Proc of the 2013 ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2013: 1181-1192

[13] Su Z, Wassermann G. The essence of command injection attacks in Web applications [C] //Proc of the 33rd ACM SIGPLAN-SIGACT Symp on Principles of Programming Languages (POPL). New York: ACM, 2006: 372-382

[14] Offensive Security. Exploits Database [EB/OL]. [2015-11-05]. https://www.exploit-db.com

[15] Halfond W G J, Orso A, Manolios P. WASP: Protecting Web applications using positive tainting and syntax-aware evaluation [J]. IEEE Trans on Software Engineering, 2008, 34(1): 65-81

[16] Medeiros I, Neves N F, Correia M. Automatic detection and correction of Web application vulnerabilities using data mining to predict false positives [C] //Proc of the 23rd Int Conf on World Wide Web. New York: ACM, 2014: 63-73

[17] Shar L K, Briand L C, Tan H K, et al. Web application vulnerability prediction using hybrid program analysis and machine learning [J]. IEEE Trans on Dependable and Secure Computing, 2015, 12(6): 688-707

[18] Shar L K, Tan H K, Briand L C. Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis [C] //Proc of the 2013 Int Conf on Software Engineering. Piscataway, NJ: IEEE, 2013: 642-651

[19] Naderi-Afooshteh A, Nguyen-Tuong A, Bagheri-Marzijarani M, et al. Joza: Hybrid taint inference for defeating Web application SQL injection attacks [C] //Proc of the 45th Annual IEEE/IFIP Int Conf on Dependable Systems and Networks(DSN). Piscataway, NJ: IEEE, 2015: 172-183

[20] Bandhakavi S, Bisht P, Madhusudan P, et al. CANDID: Preventing SQL injection attacks using dynamic candidate evaluations [C] //Proc of the 14th ACM Conf on Computer and Communications Security. New York: ACM, 2007: 12-24

[21] Sekar R. An efficient black-box technique for defeating Web application attacks [C] //Proc of the 16th Annual Network & Distributed System Security Symp. Reston, VA: Internet Society, 2009

[22] Liu A, Yuan Y, Wijesekera D, et al. SQLProb: A proxy-based architecture towards preventing SQL injection attacks [C] //Proc of the 24th Annual ACM Symp on Applied Computing. New York: ACM, 2009: 2054-2061

[23] Jovanovic N, Kruegel C, Kirda E. Pixy: A static analysis tool for detecting Web application vulnerabilities [C] //Proc of 2006 IEEE Symp on Security and Privacy. Piscataway, NJ: IEEE, 2006: 258-263

[24] Livshits V B, Lam M S. Finding security vulnerabilities in Java applications with static analysis [C] //Proc of the 14th Conf on USENIX Security Symp. Berkeley, CA: USENIX Association, 2005: 271-286

[25] Wassermann G, Su Z. Sound and precise analysis of Web applications for injection vulnerabilities [C] //Proc of the 28th ACM SIGPLAN Conf on Programming Language Design and Implementation. New York: ACM, 2007: 32-41

[26] Dahse J, Holz T. Static detection of second-order vulnerabilities in Web applications [C] //Proc of the 23rd USENIX Conf on Security Symp. Berkeley, CA: USENIX Association, 2014: 989-1003

[27] Dahse J, Holz T. Simulation of built-in PHP features for precise static code analysis [C] //Proc of the 2014 Network and Distributed System Security Symp (NDSS). Reston, VA: Internet Society, 2014

[28] Nguyen-Tuong A, Guarnieri S, Greene D, et al. Automatically hardening Web applications using precise tainting [C] //Proc of the 20th IFIP Int Information Security Conf. Berlin: Springer, 2005: 295-307

[29] Pietraszek T, Berghe C V. Defending against injection attacks through context-sensitive string evaluation [G] //LNCS 3858: Proc of the 8th Int Conf on Recent Advances in Intrusion Detection. Berlin: Springer, 2005: 124-145

[30] Xu W, Bhatkar S, Sekar R. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks [C] //Proc of the 15th Conf on USENIX Security Symp. Berkeley, CA: USENIX Association, 2006: 121-136

[31] Mui R, Frankl P. Preventing Web application injections with complementary character coding [G] //LNCS 6879: Proc of the 16th European Symp on Research in Computer Security. Berlin: Springer, 2011: 80-99

[32] Wang Yi, Li Zhoujun, Guo Tao. Literal tainting method for preventing code injection attack in Web application [J]. Journal of Computer Research and Development, 2012, 49(11): 2414-2423 (in Chinese)
(王溢, 李舟军, 郭涛. 防御代码注入式攻击的字面值污染方法[J]. 计算机研究与发展, 2012, 49(11): 2414-2423)

[33] Boyd S W, Keromytis A D. SQLrand: Preventing SQL injection attacks [G] //LNCS 3089: Proc of the 2nd Int Conf in Applied Cryptography and Network Security. Berlin: Springer, 2004: 292-302

[34] Mui R. Techniques for injection-safe Web applications [D]. New York: Polytechnic Institute of New York University, 2013

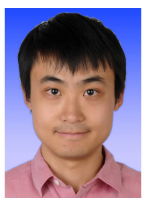
[35] Ahuja B K, Jana A, Swarnkar A, et al. On preventing SQL injection attacks [G] //AISC 395: Advanced Computing and Systems for Security. Berlin: Springer, 2016: 49-64

[36] Heart K. Process Using Universal Sanitization to Prevent Injection Attacks: USA, US20150156209A1 [P]. 2015-06-04

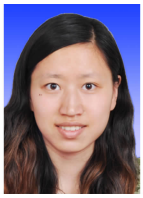
[37] Zekan B, Shtern M, Tzerpos V. Protecting Web applications via unicode extension [C] //Proc of the 22nd IEEE Int Conf on Software Analysis, Evolution and Reengineering (SANER). Piscataway, NJ: IEEE, 2015: 419-428

[38] Masri W, Sleiman S. SQLPIL: SQL injection prevention by input labeling [J]. Security and Communication Networks, 2015, 8(15): 2545-2560

Zhang Huilin, born in 1987. PhD candidate in the Institute of Computer Science and Technology, Peking University. Her main research interests include software and Web security.



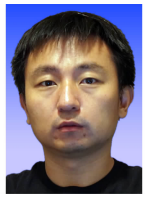
Ding Yu, born in 1988. PhD. His main research interests include software and system security (dingelish@pku.edu.cn).



Zhang Lihua, born in 1991. Master. Her main research interests include software and system security (zhanglh@pku.edu.cn).

Duan Lei, born in 1989. Master. His main research interests include software and system security (lei_duan@pku.edu.cn).

Zhang Chao, born in 1986. Postdoctor at University of California, Berkeley. His main research interests include software and system security (chaoz@berkeley.edu).



Wei Tao, born in 1975. PhD. Chief Security Scientist of Baidu Inc and the Co-organizer of UC Berkeley BitBlaze Group. His main research interests include mobile security, network and Web security (lenx.wei@gmail.com).

Li Guancheng, born in 1993. Master candidate in Peking University. His main research interests include software and system security (sunatum@outlook.com).

Han Xinhui, born in 1969. PhD. Senior engineer in Peking University. Member of China Computer Federation. His main research interests include network security and Web security.