virus BULLETIN Covering the global threat landscape

DETECTION OF VULNERABILITIES IN WEB APPLICATIONS BY VALIDATING PARAMETER INTEGRITY AND DATA FLOW GRAPHS

Abhishek Singh & Ramesh Mani
Prismo Systems, USA

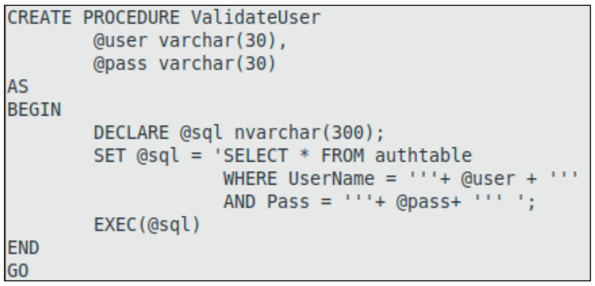
INTRODUCTION

Web application vulnerabilities are an important entry vector for threat actors. According to the 2019 *Verizon Data Breach Incident Report* [1], web applications, privilege misuse and miscellaneous errors account for 81 per cent of breaches of retail organizations.

In a paper we presented at VB2019 [2], we detailed query and parameter integrity algorithms used to detect SQL, NoSQL and OS command injection exploitation. In this follow-up paper, we detail algorithms that can be used to detect SQL injection in stored procedures, persistent cross-site scripting (XSS), and server-side request forgery (SSRF) by instrumenting web applications. Server-side request forgery allows a threat actor to access internal resources by leveraging a vulnerability in Internet-facing web applications – which can be identified by data flow analysis. SQL injection in stored procedures will lead to an additional clause in the executing SQL query, and persistent XSS in the database will lead to HTML elements in the executing query. This additional code, in the form of SQL clauses or HTML elements due to the injection-based exploitation, lays the foundation for the detection algorithms that are discussed in this paper.

TECHNICAL DETAILS OF SQL INJECTION IN STORED PROCEDURES

Like SQL queries, stored procedures can also be vulnerable to SQL injection. If the stored procedures are using dynamic SQL and the dynamic SQL query is constructed by concatenating the parameters then the stored procedures are prone to SQL injection exploitation. Figure 1 shows an example of a vulnerable stored procedure.



```
CREATE PROCEDURE ValidateUser
        @user varchar(30),
        @pass varchar(30)
AS
BEGIN
        DECLARE @sql nvarchar(300);
        SET @sql = 'SELECT * FROM authtable
                    WHERE UserName = '''+ @user + '''
                    AND Pass = '''+ @pass+ ''' ';
        EXEC(@sql)
END
GO
```

Figure 1: Stored procedure prone to SQL injection.

If the *@user* variable is *admin'*-- and the password is *none,* the query becomes:

```
'SELECT * FROM authtable WHERE UserName =
'admin'--' AND Pass = 'none' '
```

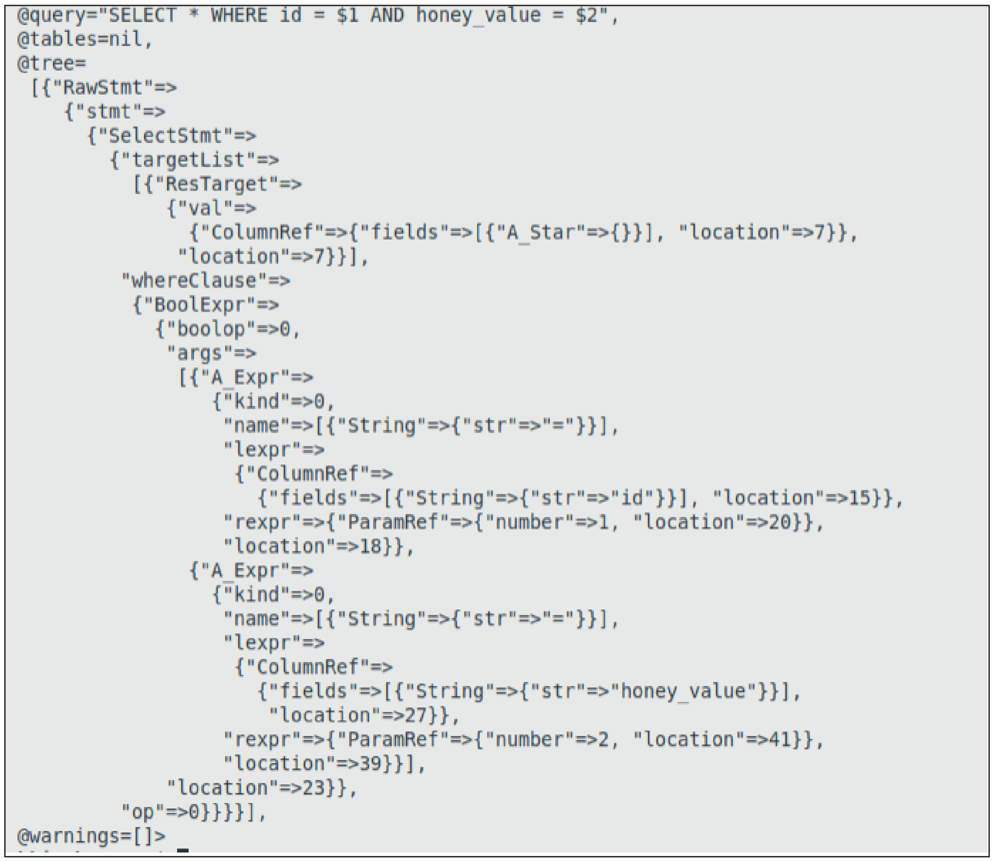
If there is a username `admin`, the query will return true and authentication will be successful.

In order to detect SQL injection in stored procedures, honey SQL queries are used. The following are some of examples of honey queries:

- `SELECT * WHERE id = '[arguments passed to Stored procedure]' AND honey_value = '2' ,`
- `(SELECT * WHERE id = '[arguments passed to Stored procedures]' AND honey_value = '1')`
- `(SELECT * WHERE id = [Arguments passed to Stored procedures] AND honey_value = '1')`
- `SELECT * WHERE id = [Arguments passed to Stored procedure] AND honey_value = '1'`
- `SELECT * WHERE id = '''[arguments passed to Stored procedure]''' AND honey_value = '2' ,`
- `(SELECT * WHERE id = '''[arguments passed to Stored procedures]''' AND honey_value = '1')`
- `'SELECT * WHERE id = "%s" AND honey_value = "2"'`

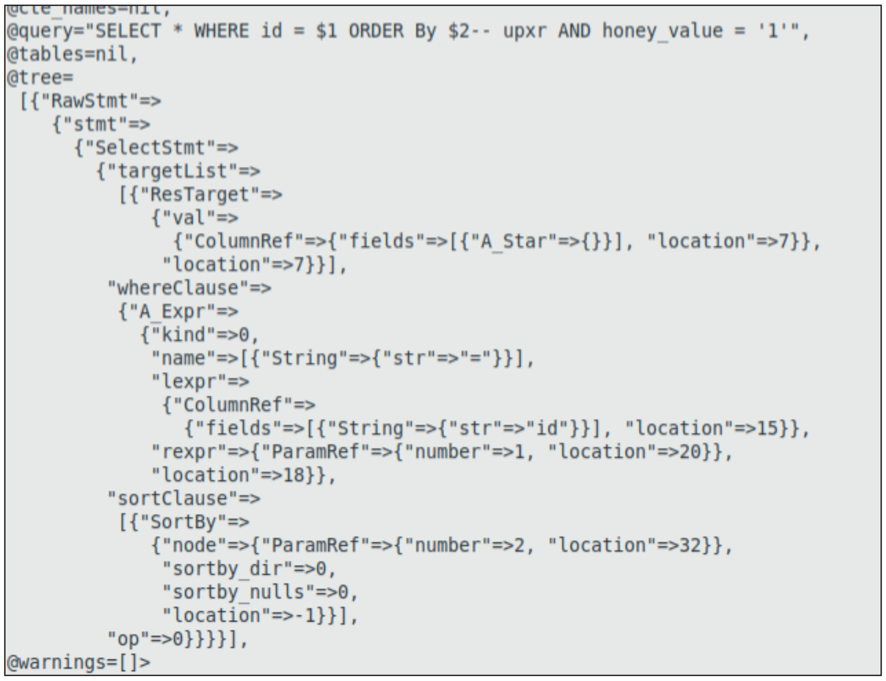
The arguments passed to the stored procedures are extracted and inserted into honey queries. If an argument is inserted in the first honey query, the normalized honey SQL query will always be `SELECT * WHERE id = $1 AND honey_value = $2` and the parse tree of the normalized first SQL query will be as shown in Figure 2. The argument of the stored procedure will always be data and not an SQL clause.

In the case of an SQL injection exploit in a stored procedure, the input parameters to the stored procedure will contain data along with the SQL clause. An example of an exploit sent to a stored procedure is: `"abhi' ORDER BY 6-- upxr"`. When the exploit is inserted in the first honey SQL query, the query becomes `SELECT * WHERE id = 'abhi' ORDER BY 6-- upxr AND honey_value = '2'`. The normalized honey query with exploit becomes `SELECT * WHERE id = $1 ORDER BY $2-- upxr AND honey_value = '2'`. The parse tree of the normalized query with SQL injection exploit is as shown in Figure 3. SQL injection exploits the stored procedures and inserts additional clauses, leading to changes in the parse tree of the normalized honey query.



```
@query="SELECT * WHERE id = $1 AND honey_value = $2",
@tables=nil,
@tree=
 [{"RawStmt"=>
    {"stmt"=>
      {"SelectStmt"=>
        {"targetList"=>
          [{"ResTarget"=>
             {"val"=>
               {"ColumnRef"=>{"fields"=>[{"A_Star"=>{}}], "location"=>7}},
              "location"=>7}}],
         "whereClause"=>
          {"BoolExpr"=>
            {"boolop"=>0,
             "args"=>
              [{"A_Expr"=>
                 {"kind"=>0,
                  "name"=>[{"String"=>{"str"=>"="}}],
                  "lexpr"=>
                   {"ColumnRef"=>
                     {"fields"=>[{"String"=>{"str"=>"id"}}], "location"=>15}},
                  "rexpr"=>{"ParamRef"=>{"number"=>1, "location"=>20}},
                  "location"=>18}},
               {"A_Expr"=>
                 {"kind"=>0,
                  "name"=>[{"String"=>{"str"=>"="}}],
                  "lexpr"=>
                   {"ColumnRef"=>
                     {"fields"=>[{"String"=>{"str"=>"honey_value"}}],
                      "location"=>27}},
                  "rexpr"=>{"ParamRef"=>{"number"=>2, "location"=>41}},
                  "location"=>39}}],
             "location"=>23}},
         "op"=>0}}}}],
@warnings=[]>
```

Figure 2: Parse tree of the normalized honey SQL query.



```
@cte_names=nil,
@query="SELECT * WHERE id = $1 ORDER By $2-- upxr AND honey_value = '1'",
@tables=nil,
@tree=
 [{"RawStmt"=>
    {"stmt"=>
      {"SelectStmt"=>
        {"targetList"=>
          [{"ResTarget"=>
             {"val"=>
               {"ColumnRef"=>{"fields"=>[{"A_Star"=>{}}], "location"=>7}},
              "location"=>7}}],
         "whereClause"=>
          {"A_Expr"=>
            {"kind"=>0,
             "name"=>[{"String"=>{"str"=>"="}}],
             "lexpr"=>
              {"ColumnRef"=>
                {"fields"=>[{"String"=>{"str"=>"id"}}], "location"=>15}},
             "rexpr"=>{"ParamRef"=>{"number"=>1, "location"=>20}},
             "location"=>18}},
         "sortClause"=>
          [{"SortBy"=>
             {"node"=>{"ParamRef"=>{"number"=>2, "location"=>32}},
              "sortby_dir"=>0,
              "sortby_nulls"=>0,
              "location"=>-1}}],
         "op"=>0}}}}],
@warnings=[]>
```

Figure 3: Parse tree of the normalized honey query with SQL injection exploit.

Honey query integrity algorithm: detection of SQL injection in stored procedures

The algorithm to detect SQL injection exploitation in stored procedures makes use of application-level hooks to instrument the functions which invoke database stored procedures, setting designated parameters and executing the stored procedures such as `prepareCall()`, `setString()` and `execute()`. The instrumentation helps to reveal the parameters that are passed to the stored procedures. The detection algorithm makes use of a set of honey queries. A parse tree of the normalized honey queries is computed and stored.

During every invocation of the API that invokes the stored procedures, designated parameters are assigned to the stored procedures and the procedures are executed; arguments passed to the stored procedures are extracted. These extracted parameters are then passed to the honey SQL queries, a parse tree of the honey queries is computed and compared with the pre-stored parse tree. If there is an additional or deleted node in the parse tree of the honey queries, an alarm for SQL injection in the stored procedures is raised.

TECHNICAL DETAILS OF SSRF

In 2019 the server-side request forgery exploitation technique [3] was used to retrieve *AWS* (*Amazon Web Services*) credentials that were subsequently used to steal the personal information of over 100 million *Capital One* customers.

In any traditional network, local host, web-based services and the internal networks are behind a firewall. SSRF allows a threat actor to exploit a vulnerability in a web application and to make an HTTP request to the local host, web-based services or in the internal networks. Figure 4 shows the vulnerable code of the *Google Forms WordPress* plug-in [4], which is prone to SSRF.

If a threat actor sends "`http://docs.google.com@internalip.com`" the request will pass the regular expression check in the code shown in Figure 4 and will be sent to the internal IP at the address denoted by "`internalip.com`". This will prompt a response from the services hosted in the internal network. As per the RFC 3986 [5], the structure of the URI will be as shown in Figure 5.

```
  foo://example.com:8042/over/there?name=ferret#nose
  \_/   \______________/\_________/ \_________/ \__/
   |           |            |            |        |
scheme     authority       path        query   fragment
   |   _____________________|__
  / \ /                        \
  urn:example:animal:ferret:nose
```

Figure 5: Structure of URI as per RFC 3986.

As per RFC 3986, the authority component is preceded by a double slash ("//") and is terminated by the next slash ("/"), question mark ("?") or number sign ("#") character, or by the end of the URI.

RFC 3986 also specifies the format of authority, as shown in Figure 6.

```
authority   = [ userinfo "@" ] host [ ":" port ]
```

Figure 6: Format of authority as per RFC 3986.

So if the exploit "`http://legitimatewebsite.com@internalip.com`" is sent to a function which parses input, such as `urllib.parse`, it will be parsed and will give the output value of the host as `legitimatewebsite.com` while `urllib.urlopen()` will show the value of the input as `internalip.com`. This mismatch in the value of the host allows a threat actor to bypass the checks in web applications. Besides the mismatch in the value of the host, RFC 3986 also specifies the option of providing host as IP-literal, IPv4Address or reg-name.

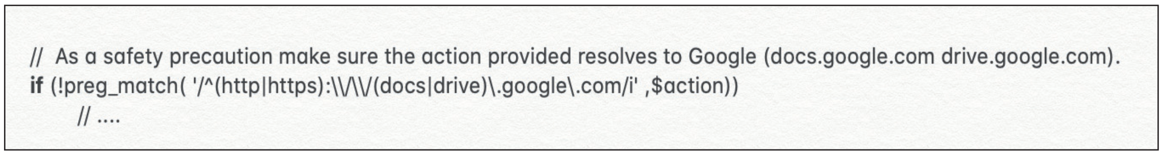
```
host        = IP-literal / IPv4address / reg-name
```

Figure 7: Options for IP address.

This means that any check for IP by a web application must ensure that the legitimate IP addresses are checked in every format.

Detection of SSRF

The algorithm to detect SSRF instruments APIs such as `urllib.urlopen()`, `urllib.request.urlopen()`, etc., which take a URI as an input parameter and open a network object denoted by the URI to read it. In addition, methods that accept user inputs, such as GET, POST, etc., are also instrumented. A program dependency graph is then used to identify the APIs that make network connections and accept



```
// As a safety precaution make sure the action provided resolves to Google (docs.google.com drive.google.com).
if (!preg_match( '/^(http|https):\/\/(docs|drive)\.google\.com/i' ,$action))
     // ....
```

Figure 4: Vulnerable check in the Google Forms plug-in, leading to SSRF.

inputs from methods that accept user inputs such as `GET()` and `POST()`. For every invocation of an API that opens a URI, a check is made to determine if the IP address of the URL to which the connection is going is either local, a loopback address, or the local link address. If the condition is found to be true, then by using the data flow graph, it can be checked whether the parameters passed to the API which opens a network object denoted by the URI are from a method which accepts external input. If this condition is found to be true, then an alert for SSRF is raised. The internal IP address as per RFC 1918 is shown in Figure 8.

The loopback IP address for most operating systems is `127.0.0.1 ~ 127.255.255.254`. If the URI is a file, then the data flow graph is used to check whether the parameters passed to the APIs which open files are from methods which accept external inputs. If the condition is found to be true then an alert for SSRF is raised.

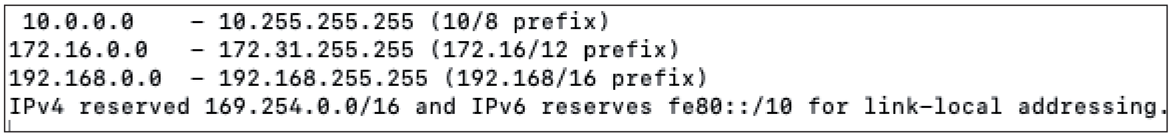
TECHNICAL DETAILS OF PERSISTENT CROSS-SITE SCRIPTING

In the case of a persistent XSS vulnerability, methods that accept user inputs, such as `POST`, etc., are used by a threat actor to inject an XSS exploit. These exploits then get stored in the backend database or in a secure store. This stored data is then sent to the victim without HTML escaping, where the XSS exploit gets executed.

Figure 9 shows exploitable code [6] for persistent XSS in a *WordPress* plug-in and the changes made to the code to remove the vulnerability. As shown in the code, there is insufficient sanitization of the plug-in version numbers before they get displayed by invoking the printf API on the corresponding plug-in page in the repository. This version number is retrieved from the database. Threat actors can exploit the vulnerability by inserting an XSS exploit with JavaScript code in the version number. This will lead to the execution of the JavaScript code. As shown in Figure 9, in the patched code the `esc_html()` function has been added to escape the string to ensure it is not passed as HTML.

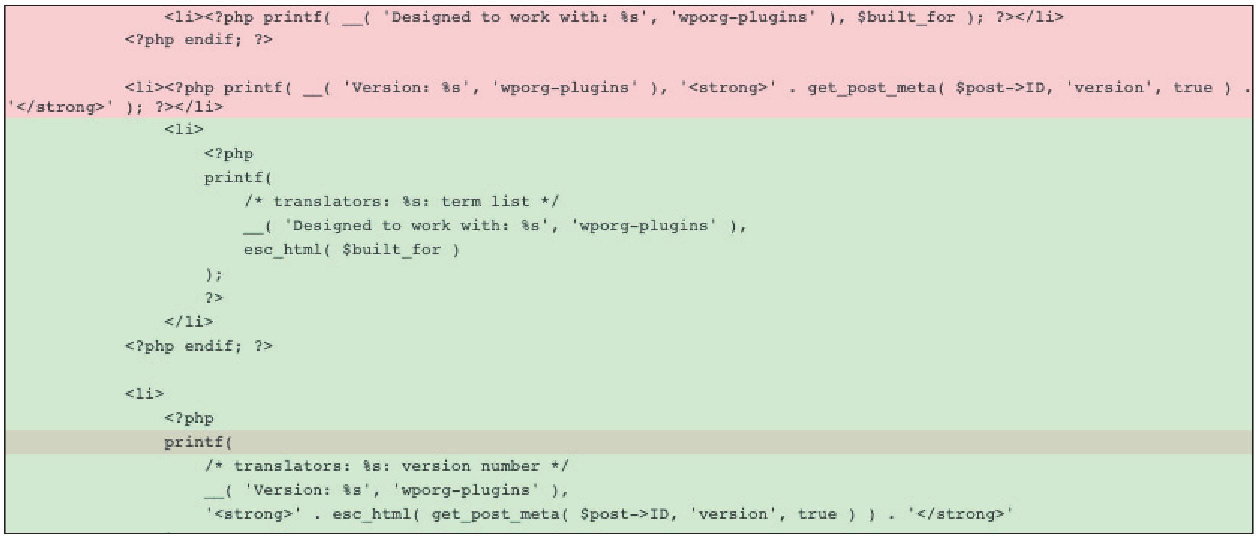
Detection of persistent cross-site scripting in databases

The algorithm to detect persistent cross-site scripting makes use of application-level hooks to construct a program dependency graph (PDG). The PDG captures the flow of data and control from methods which accept external or user input, such as GET, POST, Cookies, etc., to the functions which execute the SQL query such as `mysql_query()`, `mysql_db_query()`, `mysql_unbuffered_query()`, `pg_execute()`, `pg_query()`, `pg_query_params()`, `pg_prepare()`, `pg_send_query()`, and `pg_send_query_params()`. Once the PDG is generated, it is used to identify the legitimate SQL queries, which are the sink for the value from the methods



```
 10.0.0.0    - 10.255.255.255 (10/8 prefix)
172.16.0.0   - 172.31.255.255 (172.16/12 prefix)
192.168.0.0  - 192.168.255.255 (192.168/16 prefix)
IPv4 reserved 169.254.0.0/16 and IPv6 reserves fe80::/10 for link-local addressing.
```

Figure 8: Internal IP address as per RFC 1918.



```
                <li><?php printf( __( 'Designed to work with: %s', 'wporg-plugins' ), $built_for ); ?></li>
            <?php endif; ?>

            <li><?php printf( __( 'Version: %s', 'wporg-plugins' ), '<strong>' . get_post_meta( $post->ID, 'version', true ) .
'</strong>' ); ?></li>
                <li>
                    <?php
                    printf(
                        /* translators: %s: term list */
                        __( 'Designed to work with: %s', 'wporg-plugins' ),
                        esc_html( $built_for )
                    );
                    ?>
                </li>
            <?php endif; ?>

            <li>
                <?php
                printf(
                    /* translators: %s: version number */
                    __( 'Version: %s', 'wporg-plugins' ),
                    '<strong>' . esc_html( get_post_meta( $post->ID, 'version', true ) ) . '</strong>'
```

Figure 9: Code changes in WordPress plug-in to prevent persistent XSS.

which accept user or external input such as `GET`, `POST`, etc. For every invocation of the query execution function a data flow graph is used to check if the input to the query execution function is from user input. If the condition is found to be true, the input is parsed against the HTML parser [7] to check if the input is an HTML element. If this condition is found to be true, an alert for persistent XSS is raised.

CONCLUSION

The algorithm to detect SQL injection in stored procedures, persistent XSS, and SSRF makes use of application-level hooks. Injection-based exploitation leads to additional code, resulting in changes to the legitimate code of the application. The computation of changes in the code is carried out for every access to the database. If there is a deviation from the original code, which is identified by changes in the parse tree of honey queries or by executing the external inputs to the executing query against the HTML parser, an alert for injection exploitation is raised.

The algorithm to detect injection-based exploitation has the following inherent advantages:

- The algorithm identifies the injection vulnerability in the code during the invocation of the query execution functions. With each detected exploitation attempt, the vulnerable code path is also detected. This automatic identification of the vulnerable part of the code will help in the patching of the code, preventing further exploitation.
- The algorithm only leverages binary instrumentation of the application to detect injection-based exploitation. Hence the detection is independent of the deployment of an application and the manner in which it accepts external inputs. The application can be deployed as a backend microservice and can accept batched requests which get broken down by the middle layer and served to the rear end microservices. In this scenario the algorithm will also detect injection exploits.

The parameter and honey query integrity algorithm follows the principle of detect, respond, and remediate. Not only does the algorithm detect exploitation, but responsive measures can be applied to stop exploitation; it also provides remedial action, which will increase the exploitation complexity for a threat actor. In the case of the parameter and query integrity algorithm, remedial action is patching the vulnerable code path, which is automatically identified with each detected exploitation attempt. If the code is patched, detection alerts will decrease, increasing the exploitation complexity for a threat actor.

REFERENCES

[1] 2019 Data Breach Investigations Report. https://enterprise.verizon.com/resources/reports/2019-data-breach-investigations-report.pdf.

[2] Singh, A.; Mani, R. Catch me if you can: detection of injection exploitation by validating query and API integrity. https://www.virusbulletin.com/virusbulletin/2020/01/vb2019-paper-catch-me-if-you-can-detection-injection-exploitation-validating-query-and-api-integrity/.

[3] Krebs, B. What we can learn from the capital one hack. https://krebsonsecurity.com/tag/capital-one-breach/.

[4] Google Forms <= 0.91 - Unauthenticated Server-Side Request Forgery (SSRF). https://wpvulndb.com/vulnerabilities/9013.

[5] Uniform Resource Identifier (URI): Generic Syntax RFC 3986. https://tools.ietf.org/html/rfc3986.

[6] Code changes to remove persistent XSS in WordPress. https://meta.trac.wordpress.org/changeset/7195.

[7] HTML parser. https://godoc.org/golang.org/x/net/html.

Head of Testing: Peter Karsai

Security Test Engineers: Gyula Hachbold, Adrian Luca, Csaba Mészáros, Tony Oliveira, Ionuţ Răileanu

Sales Executive: Allison Sketchley

Editorial Assistant: Helen Martin

© 2020 Virus Bulletin Ltd, Manor House - Office 6, Howbery Business Park, Wallingford OX10 8BA, UK

Tel: +44 20 3920 6348 Email: editorial@virusbulletin.com

Web: https://www.virusbulletin.com/