TEKNIK PROTEKSI TERHADAP SERANGAN SQL *INJECTION* MENGGUNAKAN KONSEP AMNESIA PADA APLIKASI *WEB*

Syah Askar Akbar
Program Studi Teknik Informatika, Fakultas Ilmu Komputer
Universitas Dian Nuswantoro, Semarang 50131
E-mail : syah_askar_akbar@yahoo.co.id
syah.askar.akbar@gmail.com

Ajib Susanto
Program Studi Teknik Informatika, Fakultas Ilmu Komputer
Universitas Dian Nuswantoro, Semarang 50131
E-mail : a71b@dosen.dinus.ac.id;
ajibsusanto@gmail.com

ABSTRAK

Teknologi web telah berkembang demikian pesat hingga saat ini. Akses internet pun semakin mudah didapat, dengan biaya yang relatif terjangkau. Oleh karena itu, rasanya tidak ada alasan untuk tidak menjadikan aplikasi web sebagai penunjang aktivitas manusia. Namun, sebagai aplikasi berbasis jaringan, aplikasi web tentu sangat rentan terhadap serangan.

Berbagai jenis serangan telah diteliti, dan ditemukan bahwa SQL injection berada pada peringkat pertama dari sepuluh jenis serangan paling berbahaya yang sering terjadi pada tahun 2007 s.d. 2010. Di sisi lain, berdasarkan hasil penelitian di tahun 2010, terdapat kurang lebih 23 macam teknik proteksi terhadap serangan SQL injection telah diciptakan oleh para peneliti IEEE Computer Society.

Sebagai wujud kepedulian, penulis telah merealisasikan salah satu konsep proteksi yang cukup populer di kalangan para peneliti, yakni konsep AMNESIA. Konsep ini sebenarnya telah direalisasikan ke dalam konteks bahasa Java oleh penciptanya. Namun, penulis merasa perlu untuk memodifikasi konsep tersebut ke dalam konteks bahasa PHP. Hal ini dilakukan karena, pada kenyataannya aplikasi web yang ada hingga saat ini mayoritas berbasis PHP.

Hasil penelitian ini, berupa tool proteksi yang mampu mendeteksi dan mencegah serangan SQL injection. Penulis berharap, tool tersebut dapat menjadi salah satu alternatif untuk meminimalisir serangan SQL injection yang terjadi pada aplikasi web berbasis PHP.

Kata kunci: SQL *injection*, AMNESIA, Java, PHP

1. PENDAHULUAN

Perkembangan aplikasi *web* yang semakin canggih seolah mampu menjadikan dunia ini tidak terbatas ruang maupun waktu. Berkat dukungannya dalam penyampaian informasi yang cepat dan akurat, banyak pihak yang mulai mengadopsi aplikasi *web* ke dalam dunia mereka. Namun, sebagaimana yang dikemukakan oleh Meike dkk. [1], aplikasi *web* ternyata jauh lebih rentan terhadap ancaman keamanan daripada aplikasi *desktop* konvensional yang berdiri sendiri. Hal ini disebabkan karena aplikasi *web* merupakan aplikasi berbasis jaringan *internet*, sehingga siapa pun dapat mengaksesnya dari mana pun dan kapan pun.

Serangan-serangan yang mengancam keamanan aplikasi *web* cukup beragam, seperti: XSS (*Cross Site Scripting*), DoS (*Denial of Service*), SQL (*Structured Query Language*) *injection*, dan lain-lain. Berdasarkan hasil penelitian yang dikemukakan oleh OWASP (*Open Web Application Security Project*) [2], serangan SQL *injection* termasuk ke dalam sepuluh besar teknik penyerangan paling berbahaya terhadap aplikasi *web*. Hasil penelitian pada tahun 2004, menunjukkan bahwa serangan SQL *injection* menempati posisi ke-6, dan naik ke posisi ke-2 pada tahun 2006, kemudian antara tahun 2007 s.d. 2010 serangan SQL *injection* kembali naik hingga berada pada posisi pertama.

Menanggapi persoalan ini, para peneliti yang tergabung dalam IEEE (*Institute of Electrical and Electronics Engineers*) *Computer Society* telah berusaha menghadirkan berbagai metode untuk mengatasi serangan SQL *injection*. Tajpour dan Shooshtari [3] telah melakukan evaluasi terhadap 23 macam metode perlindungan dari serangan SQL *injection*. Selain itu, ternyata masih ada lagi publikasi lain yang muncul setelah evaluasi tersebut, seperti: Halder dan Agostino [4] dengan pendekatan analisis *obfuscation-based*, Balasundaram dan Ramaraj [5] dengan metode "*X-Log Authentication Technique*", Sangita dkk. [6] dengan metode *filter* permintaan HTTP (*HyperText Transfer Protocol*) berdasakan *signature*, Alserhani dkk. [7] dengan metode "*Event-Based Alert Correlation System*", Pratap dkk. [8] dengan metode "*Font Level Tainting*", Jiao dkk. [9] dengan metode "*High-Interaction Honeypot System*", Rawat dkk. [10] yang mencoba menghadirkan standar-standar pengodean dan pedoman praktis keamanan, Kai-Xiang dkk. [11] dengan *tool* TransSQL-nya, Telang dkk. [12] dengan metode yang mengadopsi algoritma Hirschberg, serta Rawat dan Shrivastav [13] dengan *tool SVM*-nya. Jika melihat berbagai usaha yang dilakukan oleh mereka, dapat dibayangkan bahwa serangan SQL *injection* bukanlah sekedar 'AND 1=1', melainkan persoalan yang jauh lebih kompleks daripada itu.

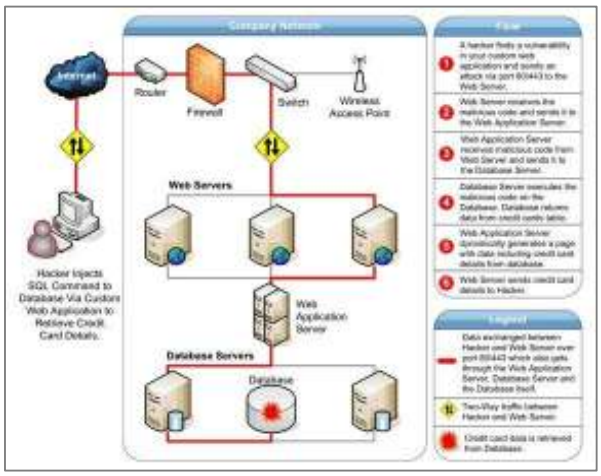
Dalam kegiatan penelitian ini, penulis memilih untuk menggunakan konsep proteksi AMNESIA (*Analysis and Monitoring for NEutralizing SQL Injection Attacks*), yang dikemukakan oleh Halfond dan Alessandro [14, 15]. Meski bukan teknik baru, namun teknik ini diklaim sangat efektif dan efisien, serta cukup mudah untuk dimengerti dan diimplementasikan. Konsep proteksi yang mereka ajukan menggunakan pendekatan berbasis model, yakni membandingkan model dari dua struktur *query* SQL yang berbeda. Model pertama diambil dari aplikasi *web* itu sendiri, sedangkan model yang kedua diambil dari hasil *input* yang diberikan oleh *user*. Selanjutnya, kedua model tersebut dibandingkan, apabila kedua model tersebut memiliki struktur yang sama, maka anggapannya adalah bukan merupakan serangan SQL *injection*, namun jika sebaliknya, maka akan dianggap sebagai serangan SQL *injection*.

Di sisi lain, konsep AMNESIA ternyata khusus diciptakan untuk aplikasi-aplikasi *web* berbasis Java. Padahal, hasil *survey* Wappalyzer [16] menunjukkan bahwa, dari sekian banyak bahasa pemprograman *web*, ternyata 97% aplikasi *web* yang ada hingga saat ini telah menggunakan bahasa pemprograman PHP (PHP: *Hypertext Preprocessor*) sebagai basisnya. Oleh karena itu, penulis telah memutuskan untuk memperluas konsep tersebut agar dapat diterapkan ke dalam aplikasi-aplikasi *web* berbasis PHP. Dengan demikian, hasil penelitian ini diharapkan mampu meminimalisir serangan SQL *injection* yang mengancam keamanan aplikasi-aplikasi *web* berbasis PHP.

2. LANDASAN TEORI

2.1 Serangan SQL *Injection*

Serangan SQL *injection* biasa dikenal dengan istilah SQLIA (*Structured Query Language Injection Attacks*). Menurut OWASP [2], SQLIA termasuk ke dalam kategori *Injection Flaws*, yakni injeksi data ilegal ke dalam *interpreter*, dengan cara menyamar sebagai *query* atau *command* legal.



Gambar 1: Ilustrasi SQLIA yang dilakukan oleh *hacker* [Sumber: 21].

Gambar 1 memperlihatkan ilustrasi sederhana mengenai SQLIA. Gambar tersebut memperlihatkan mekanisme SQLIA yang dilakukan oleh *hacker* yang ingin mengorek informasi tentang rincian kartu kredit pelanggan. Definisi SQLIA sendiri menurut Clarke [19] adalah salah satu jenis celah keamanan, dimana penyerang memiliki keleluasaan untuk memanipulasi *query-query* SQL yang disampaikan oleh aplikasi *web* kepada basisdata. Dengan keleluasaan tersebut, penyerang mampu memanfaatkan *syntax* dan kemampuan dari SQL itu sendiri untuk mendapatkan informasi yang diperlukan.

SQLIA memiliki beberapa karakteristik. Halfond dkk. [20] mengelompokkan karakteristik SQLIA berdasarkan dua sudut pandang berbeda, yaitu: mekanisme injeksi dan tujuan penyerangan. Berdasarkan mekanisme injeksi, SQLIA memiliki karakteristik sebagai berikut:

a) Injeksi melalui *user input*: Penyerang memanfaatkan *field-field input* yang terdapat dalam sebuah *form* aplikasi *web*, seperti: *login*, *contact*, dll. *Form* tersebut mengirimkan variabel-variabel yang menampung nilai *input* melalui permintaan HTTP GET dan POST.
b) Injeksi melalui *cookie*: *Cookie* adalah tempat penyimpanan informasi sementara yang terdapat pada sisi klien. *Cookie* biasanya memuat informasi *state* yang dibangkitkan oleh aplikasi *web*. Informasi ini berguna untuk mempercepat *loading* halaman *web*, karena pada saat *user* mengunjungi kembali halaman yang sama, aplikasi *web* tidak perlu lagi mengambil data ke *server*. Apabila aplikasi *web* menggunakan *cookie* untuk menyelenggarakan suatu operasi *query* SQL, maka tentunya kondisi ini akan sangat menguntungkan bagi penyerang. Dengan memanipulasi perintah-perintah SQL tertentu berdasarkan *cookie* yang ada, penyerang mampu mengorek informasi-informasi yang diperlukan dari dalam basisdata.
c) Injeksi melalui variabel *server*: Variabel *server* adalah sekumpulan variabel yang memuat informasi HTTP, *network header*, dan sejenisnya. Aplikasi *web* menggunakan variabel *server* untuk berbagai kepentingan, seperti sistem pelaporan statisik penggunaan *bandwidth*, *browser*, dll. Apabila variabel tersebut disimpan ke dalam basisdata tanpa proses sanitasi, tentunya akan menimbulkan celah keamanan yang rentan terhadap SQLIA.
d) Injeksi *second-order*: Dalam injeksi *second-order*, penyerang mempelajari mekanisme suatu *input* hingga mampu mengetahui pada bagian-bagian mana dari mekanisme tersebut dapat disisipkan perintah-perintah ilegal tanpa terdeteksi. Sebagai contoh, penyerang ingin mengganti *password* korban tanpa mengetahui *password* korban yang bersangkutan. Dalam hal ini, penyerang harus mengetahui lebih dulu *username* dari si korban, dan memahami mekanisme penggantian *password* dalam sistem yang bersangkutan. Untuk memahami mekanisme tersebut, penyerang tentunya harus memiliki sistem yang sama dengan yang digunakan oleh si korban. Andaikan si penyerang telah mengetahui mekanisme penggantian *password*, dan mampu memperkirakan struktur *query*-nya seperti berikut:

```
<%
...
queryString = "UPDATE users SET password='" + newPassword + "' WHERE
                                username='" + userName + "' AND
                                password='" + oldPassword + "'";
...
%>
```

Gambar 2: Perkiraan struktur *query* dari mekanisme penggantian *password* [Sumber: 20].

```
<%
...
queryString = "UPDATE users SET password='newpwd' WHERE
                                username='admin'--' AND
                                password='oldpwd'";
...
%>
```

Gambar 3: Kondisi struktur *query* setelah dimanipulasi oleh penyerang [Sumber: 20].

Variabel newPassword adalah untuk menampung nilai *password* baru, oldPassword adalah untuk menampung nilai *password* sebelumnya, dan userName adalah untuk menampung nilai *username* si korban. Misalkan *username* si korban adalah 'admin' (tanpa tanda kutip), dan si penyerang akan melakukan manipulasi dengan cara memasukkan *field username* dengan nilai admin'-- dan *field new password* dengan nilai newpwd. Penyerang tidak perlu mengetahui *password* si korban, namun untuk memperjelas ilustrasi, penulis akan menggunakan oldpwd sebagai perwakilan dari oldPassword, sehingga kondisi *query* yang sekarang terlihat seperti pada Gambar 3. Dalam SQL, simbol -- menunjukkan awal dari komentar, sehingga *string* atau karakter yang terletak setelah simbol tersebut akan dianggap sebagai komentar, dan tentunya tidak akan dieksekusi. Berdasarkan kondisi ini, maka dapat dipastikan bahwa *password* si korban dapat diubah oleh si penyerang tanpa menghiraukan keberadaan *password* sebelumnya.

Sedangkan berdasarkan kategori tujuan penyerangan, SQLIA memiliki sepuluh karakteristik, diantaranya adalah sebagai berikut:

a) *Identifying injectable parameters*: ingin menyelidiki suatu aplikasi *web* untuk menemukan parameter-parameter atau *field-field input user* yang rentan terhadap SQLIA.
b) *Performing database finger-printing*: ingin mencari tahu tentang jenis dan versi dari basisdata yang digunakan oleh suatu aplikasi *web*, yang berguna untuk menentukan susunan manipulasi *query* atau *command* penyerangan yang cocok untuk diterapkan kepadanya.
c) *Determining database schema*: untuk mengekstrak data-data dari dalam basisdata secara lebih akurat. Penyerang mengincar informasi dari basisdata *schema* karena di dalamnya terdapat informasi-informasi tentang struktur basisdata secara keseluruhan, seperti: nama-nama tabel, nama-nama kolom, tipe data kolom, dll.

d) *Extracting data*: jenis SQLIA pada umumnya bertujuan untuk hal yang satu ini, yakni mengekstrak data berdasarkan keinginan dari si penyerang. Penyerangan ini biasanya bergantung pada aplikasi *web* yang menjadi objek. Misalnya, pada aplikasi *web* toko *online*, kebanyakan penyerang ingin mengekstrak data tentang rincian kartu kredit yang tersimpan di dalam basisdata milik aplikasi *web* tersebut.
e) *Adding or modifying data*: tujuan penyerangan adalah menambah atau mengubah informasi yang ada di dalam basisdata.
f) *Performing denial of service*: serangan ini dilakukan untuk menon-aktifkan koneksi basisdata dari suatu aplikasi *web*, yang mengakibatkan terputusnya layanan koneksi ke *user*.
g) *Evading detection*: biasanya penyerang ingin menguji suatu sistem perlindungan dari suatu aplikasi *web*. Penyerang dengan berbagai cara berusaha agar serangannya dapat lolos dari mekanisme pemeriksaan dan pendeteksian yang ada.
h) *Bypassing authentication*: tujuan penyerangan ini adalah melompati mekanisme otentikasi yang membentengi basisdata dan aplikasi *web*.
i) *Executing remote commands*: memanfaatkan perintah-perintah yang tersimpan di dalam basisdata target. Biasanya perintah-perintah tersebut berupa fungsi atau prosedur tersimpan (*stored procedure*) yang tersedia di dalam basisdata tersebut.
j) *Performing privilege escalation*: mengeksploitasi hak-hak akses pengguna (*user privileges*) terhadap basisdata. Berlawanan dengan *Bypassing authentication*, *performing privilege escalation* lebih memilih untuk menghadapi otentikasi (tidak melompatinya), karena si penyerang telah memodifikasi hak aksesnya ke posisi yang jauh lebih tinggi dari sebelumnya.

SQLIA sangat beragam jenisnya, sehingga rasanya cukup sulit untuk mengelompokkan seluruhnya ke dalam sebuah klasifikasi yang atomik. Akan tetapi, Halfond dkk. [20] telah mencoba untuk mengelompokkannya ke dalam tujuh golongan. Meski tidak semua jenis SQLIA dapat masuk ke dalam daftar klasifikasi tersebut, namun setidaknya klasifikasi tersebut dapat memberikan gambaran sederhana mengenai kompleksitas SQLIA. Berikut ini adalah jenis-jenis SQLIA tersebut:

a) *Tautologies*
Tujuan penyerangannya adalah: *bypassing authentication*, *identifying injectable parameters*, dan *extracting data*. Umumnya SQLIA yang berbasis tautologi melakukan injeksi dengan satu atau lebih pernyataan kondisional, sedemikian rupa hingga hasil injeksi tersebut akan selalu bernilai benar.
Sebagai contoh, misalkan ada sebuah *form login* yang mengandung tiga buah *field input*, yakni *username*, *password*, dan *token*. Struktur *query* untuk *form* tersebut diperlihatkan pada Gambar 4.

```
SELECT accounts FROM users WHERE username='varUserName' AND
                                password='varPassWord' AND
                                token='varToken';
```

Gambar 4: Struktur *query* dalam mekanisme *login* sebelum diterapkan SQLIA berbasis tautologi [Sumber: 20].

```
SELECT accounts FROM users WHERE username='' OR 1=1 --' AND
                                password='' AND
                                token='';
```

Gambar 5: Struktur *query* dalam mekanisme *login* setelah diterapkan SQLIA berbasis tautologi [Sumber: 20].

Apabila penyerang memasukkan data username dengan nilai ‘ OR 1=1 --, tanpa memasukkan nilai apapun ke dalam *field password* maupun *token*, maka kondisi struktur *query*-nya menjadi seperti terlihat pada Gambar 5. Karena simbol -- menandakan sebuah komentar, maka *string* setelah simbol tersebut tidak akan dieksekusi. Dengan demikian hasil eksekusi *query* tersebut akan selalu bernilai benar atau bersifat tautologi.

b) *Illegal (Logically Incorrect) Queries*
Tujuan penyerangannya adalah: *identifying injectable parameters*, *performing database finger-printing*, dan *extracting data*. Keberhasilan serangan ini akan memberikan informasi penting kepada si penyerang mengenai jenis dan struktur dari basisdata yang digunakan oleh aplikasi *web*. Celah keamanan yang dapat dimanfaatkan adalah halaman kesalahan yang dibangkitkan oleh *server* aplikasi *web*, dimana halaman tersebut memuat informasi-informasi penting mengenai struktur basisdata.
Ketika serangan ini dilancarkan, penyerang menginjeksi pernyataan-pernyataan tertentu agar basisdata memberikan informasi mengenai kesalahan *syntax*, tipe data, ataupun logika. Kesalahan *syntax* dapat digunakan untuk mengidentifikasi parameter *injectable*. Kesalahan tipe data digunakan untuk menyimpulkan dengan akurat mengenai tipe data dari suatu kolom atau untuk mengekstrak data. Sedangkan kesalahan logika biasanya digunakan untuk mengetahui nama tabel dan kolom yang menyebabkan munculnya kesalahan tersebut.

```
SELECT accounts FROM users WHERE username='' AND
                                password='' AND
                                token='' convert(int,(select
                                top 1 name from sysobjects where
                                xtype='u'))--';
```

Gambar 6: Struktur *query* dalam mekanisme *login* saat serangan SQLIA berbasis *illegal (logically incorrect) queries* [Sumber: 20].

```
Microsoft OLE DB Provider for SQL Server (0x80040E07) Error
converting nvarchar value 'CreditCards' to a column of data
type int.
```

Gambar 7: Pesan kesalahan yang muncul setelah *query* pada Gambar 6 dieksekusi [Sumber: 20]

Sebagai contoh, misalkan penyerang ingin mengetahui nama-nama tabel beserta strukturnya dari dalam basisdata milik aplikasi *web* pada contoh sebelumnya (Gambar 4). Untuk melakukan hal ini, penyerang menginjeksi pernyataan ’ convert(int,(select top 1 name from sysobjects where xtype=’u’))-- ke dalam *field token*, sehingga struktur *query*-nya menjadi terlihat seperti pada Gambar 6. Dalam contoh ini, penyerang memperkirakan bahwa tipe dan versi basisdata yang digunakan oleh aplikasi *web* adalah Microsoft SQL Server.
Penginjeksian kode atau pernyataan seperti terlihat pada Gambar 2.6, bertujuan untuk mengekstrak nama tabel *user* pertama yang diperoleh dari tabel metadata (sysobjects) menggunakan kode xtype=’u’. Pernyataan tersebut bermaksud untuk mengonversi nama tabel ke dalam tipe data *integer*, namun tentu saja operasi konversi tersebut tidak dapat dilakukan oleh basisdata, karena tidak mengikuti aturan yang berlaku. Dengan demikian, basisdata akan menampilkan pesan kesalahan seperti ditunjukkan pada Gambar 7.

Berdasarkan pesan kesalahan tersebut, penyerang dapat mengetahui dua hal penting, yaitu: tipe basis data yang digunakan (SQL Server), dan nama tabel pertama milik *user* (CreditCards). Penyerang dapat menggunakan pola yang sama untuk mengorek informasi lebih dalam mengenai struktur tabel tersebut, seperti nama-nama kolom yang ada beserta tipe datanya, dan bahkan dapat mengorek semua informasi basisdata dalam *server* tersebut.

c) *Union Query*

Tujuan penyerangannya adalah: *bypassing authentication*, dan *extracting data*. Jenis serangan seperti ini biasanya menggunakan injeksi pernyataan dalam bentuk: UNION SELECT <parameter injeksi query>. Sebagai contoh, andaikan pada langkah sebelumnya si penyerang berhasil memperoleh informasi mengenai struktur dari tabel CreditCards, yakni mengetahui salah satu isi kolom acctNo dan nama kolom cardNo (belum tahu isinya).

Kemudian si penyerang ingin mengetahui nomor kartu kredit dari akun yang memiliki acctNo=10032. Untuk memperoleh informasi tersebut, penyerang menginjeksi pernyataan *query* ' UNION SELECT cardNo from CreditCards where acctNo=10032 -- ke dalam *field username* seperti terlihat pada Gambar 8.

```
SELECT accounts FROM users WHERE username='' UNION SELECT cardNo
                from CreditCards where acctNo=10032 --' AND
                password='' AND
                token='';
```

Gambar 8: Tampilan struktur *query* dalam mekanisme *login* saat penerapan SQLIA berbasis *union query* [Sumber: 20].

```
SELECT accounts FROM users WHERE username='doe' AND
                        password=''; drop table users --' AND
                        token='123';
```

Gambar 9: Tampilan struktur *query* dalam mekanisme *login* saat penerapan SQLIA berbasis *piggy-backed queries* [Sumber: 20].

Hasil akhir yang diperoleh dari eksekusi *query* tersebut adalah gabungan dari pernyataan pertama dan kedua, dimana pernyataan pertama menghasilkan *null*, sedangkan pernyataan kedua menghasilkan nomor kartu kredit.

d) *Piggy-Backed Queries*

Tujuan penyerangannya adalah: *extracting data*, *adding or modifying data*, *performing denial of service*, dan *executing remote commands*. Ciri khas dari penyerangan ini adalah penggunaan operator *delimiter* (;), karena biasanya penyerang menyisipkan *query* injeksi mengikuti *query valid* yang diletakkan setelah operator tersebut. Sebagai contoh, misalkan penyerang ingin menghapus tabel users pada contoh sebelumnya. Untuk melakukan hal ini, penyerang menginjeksi *query* '; drop table users -- ke dalam *field password*, seperti terlihat pada Gambar 9.

Hasil dari penyerangan tersebut adalah terhapusnya tabel users dari dalam basisdata. Cara ini juga dapat digunakan untuk menambahkan suatu nilai ke dalam basis data, dan juga mampu mengeksekusi *stored procedure* yang ada.

e) *Stored Procedures*

Tujuan penyerangannya adalah: *performing privilege escalation*, *performing denial of service*, dan *executing remote commands*. Serangan ini mencoba untuk mengeksekusi *stored procedure* yang ada di dalam basisdata. *Stored procedure* merupakan perluasan fungsionalitas basisdata dan mampu berinteraksi dengan sistem operasi.

Oleh karena itu, apabila si penyerang mampu mengeksekusi *stored procedure* yang ada, maka kemungkinan besar si penyerang juga dapat berinteraksi dengan sistem operasi yang digunakan oleh aplikasi *web*. Sebagai contoh, misalkan penyerang ingin mematikan *server* basisdata. Apabila mekanisme *login* pada contoh-contoh sebelumnya dimodifikasi ke dalam bentuk *stored procedure*, maka akan terlihat seperti pada Gambar 10.

```
CREATE PROCEDURE DBO.isAuthenticated
    @varUserName varchar2, @varPassWord varchar2, @varToken int
AS
    EXEC("SELECT accounts FROM users WHERE
                            username='" +@varUserName+ "' and
                            password='" +@varPassWord+ "' and
                            token='" +@varToken+ "');
GO
```

Gambar 10: Tampilan *strored procedure* hasil modifikasi mekanisme *login* pada contoh-contoh sebelumnya [Sumber: 20].

```
SELECT accounts FROM users WHERE username='doe' AND
                            password=''; SHUTDOWN; --' AND
                            token='123'
```

Gambar 11: Struktur *query* dalam mekanisme *login* saat penerapan SQLIA berbasis *stored procedure* [Sumber: 20].

Untuk mematikan *server* basisdata, penyerang cukup menginjeksi pernyataan '; SHUTDOWN; -- ke dalam *field username* atau *password*, seperti terlihat pada Gambar 11. Jika *query* tersebut dieksekusi dalam sebuah *stored procedure* seperti pada Gambar 10, maka koneksi ke *server* basisdata akan terputus dengan seketika.

f) *Inference*

Tujuan penyerangannya adalah: *identifying injectable parameters*, *extracting data*, dan *determining database schema*. Serangan semacam ini biasanya dilakukan apabila serangan-serangan seperti yang terdapat pada penjelasan sebelumnya tidak berhasil dilakukan. Jenis serangan ini adalah berupa interaksi tanya-jawab antara si penyerang dengan *server* basisdata. Informasi diperoleh lewat jawaban "ya" atau "tidak" sesuai tanggapan yang diberikan oleh *server* basisdata. Jawaban "ya", analog dengan tampilan halaman *web* normal (tanpa cacat atau tanpa pesan kesalahan), sedangkan jawaban "tidak", analog dengan tampilan yang cacat atau munculnya pesan kesalahan. Namun, dapat juga menjadi sebaliknya, tergantung dari pola penyerangan yang digunakan oleh si penyerang.

Jenis penyerangan ini terbagi menjadi dua kategori, yaitu: *blind injection*, dan *timing attacks*. Perbedaannya hanya terletak pada penggunaan waktu *delay*. *Blind injection* tidak menggunakan waktu *delay* untuk menunggu jawaban dari *server*, sedangkan *timing attacks* menggunakan *delay* dalam menyimpulkan jawaban dari *server*.

```
SELECT accounts FROM users WHERE username='legalUser' and 1=0 --'
                        AND password='' AND token='';

SELECT accounts FROM users WHERE username='legalUser' and 1=1 --'
                        AND password='' AND token='';
```

Gambar 12: Tampilan dua contoh struktur *query* dalam mekanisme *login* untuk memulai SQLIA berbasis *inference* (*blind injection*) [Sumber: 20].

Gambar 13: Tampilan struktur *query* dalam mekanisme *login* saat penerapan SQLIA berbasis *inference* (*timing attacks*) [Sumber: 20].

Gambar 12 menampilkan dua buah struktur *query* yang digunakan untuk memulai SQLIA berbasis *inference* (*blind injection*). Kedua struktur tersebut sebenarnya sama, hanya saja berbeda dalam hal nilai *input* yang diinjeksi. Pada struktur yang pertama, yakni pada baris ke-1 s.d. ke-2, penyerang mencoba masuk dengan *username* yang legal, namun diikuti dengan pernyataan injeksi, yaitu: legalUser' and 1=0 --. Sedangkan pada struktur yang kedua, yakni pada baris ke-4 dan ke-5, dimasukkan pernyataan: legalUser' and 1=1 --.

Untuk mengidentifikasi apakah *form login* rentan terhadap SQLIA, maka dilakukan uji-coba menggunakan kedua nilai *input* tersebut. Pertama, apabila struktur *query* pertama (baris ke-1 s.d. ke-2) dieksekusi, seharusnya akan selalu menghasilkan nilai yang selalu salah atau *false*. Dengan demikian, apabila sistem keamanannya bagus, maka seharusnya aplikasi *web* menampilkan pesan kesalahan setelah *query* tersebut dieksekusi. Sebaliknya, apabila tidak ada pesan kesalahan yang ditampilkan (berjalan normal tanpa tanggapan sedikit pun), maka dianggap bahwa aplikasi *web* tersebut rentan terhadap SQLIA.

Untuk meyakinkan penyerang mengenai hasil awal penyerangan tersebut, maka dimanfaatkan struktur *query* yang kedua (baris ke-4 dan ke-5). Apabila struktur *query* tersebut dieksekusi, seharusnya akan menghasilkan nilai yang selalu benar atau *true*. Dalam hal ini, apabila muncul pesan kesalahan, maka dianggap aplikasi *web* rentan terhadap SQLIA. Namun sebaliknya, aplikasi *web* aman dari SQLIA.

Langkah selanjutnya adalah mengekstrak nama tabel dari dalam basisdata dengan memanfaatkan informasi yang disimpan dalam tabel sysobjects (misal penyerang sudah mengetahui jenis basisdata yang digunakan). Untuk melakukan hal ini, digunakan teknik *timing attacks*, yakni dengan menginjeksi *query* legalUser' and ASCII(SUBSTRING((select top 1 name from sysobjects),1,1)) > X WAITFOR 5 -- ke dalam *filed username*, seperti terlihat pada Gambar 13.

Serangan tersebut mencoba untuk mengekstrak satu per satu huruf yang membentuk nama tabel. Struktur *query* pada Gambar 13 mencoba untuk menanyakan kebenaran dari huruf pertama nama tabel, yakni apakah lebih besar dari karakter X atau sebaliknya. Benar atau tidaknya jawaban dari *server* basisdata, dapat dilihat dari reaksi yang terjadi pada halaman *web* dalam waktu lima detik setelah *query* tersebut dikirim.

Huruf-huruf selanjutnya dapat diketahui dengan cara yang sama, hanya tinggal menggeser *pointer* karakter pada fungsi SUBSTRING, dan mengubah-ubah karakter uji-coba (selain X). Karena prosesnya yang memakan waktu lama, umumnya serangan ini dijadikan sebagai senjata terakhir oleh penyerang dalam konteks SQLIA.

g) *Alternate Encodings*

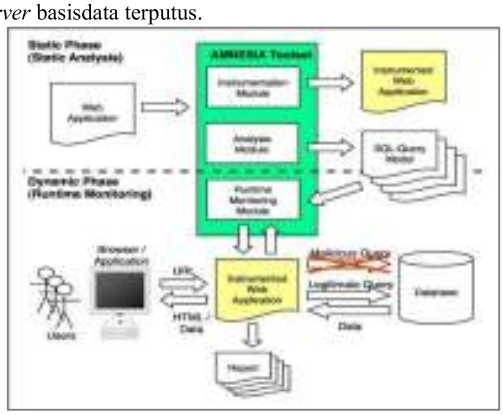
Tujuan utama penyerangan ini adalah *evading detection*. Apabila aplikasi *web* menggunakan pengamanan terhadap SQLIA, yang mampu menangkal teknik-teknik penyerangan seperti pada penjelasan sebelumnya, maka *alternate encodings* adalah pilihan tepat untuk menggantikannya. Teknik ini memanfaatkan karakter *encoding* heksadesimal, ASCII, dan *Unicode*, untuk menghindari pendeteksian yang dilakukan oleh pengamanan aplikasi *web* berbasis *character escaping*.

Agar *query* injeksi mampu menghindari pendeteksian (yang biasanya memeriksa karakter-karakter terlarang), maka *query* tersebut harus dikonversi terlebih dahulu ke dalam kode-kode yang tidak terdeteksi oleh pengamanan tersebut. Sebagai contoh, misalkan mekanisme *login* yang ada pada Gambar 2.10 telah menggunakan pengamanan, sedemikian rupa sehingga karakter-karakter terlarang seperti kutip tunggal (‘), operator komentar SQL (--), dan lain-lain, tidak dapat masuk ke dalam struktur *query* yang *valid*.

Untuk melewati pengamanan tersebut, maka penyerang harus mengonversi *string* SHUTDOWN ke dalam basis kode heksadesimal 0x73687574646f776e, dan kemudian menginjeksinya melalui *field username* menggunakan pernyataan legalUser’; exec(char(0x73687574646f776e)) --, seperti terlihat pada Gambar 14.

Gambar 14: Struktur *query* dalam mekanisme *login* saat penerapan SQLIA berbasis *alternate encodings* [Sumber: 20].

Fungsi char() berguna untuk mengonversi kode heksadesimal menjadi serangkaian karakter ASCII saat sampai ke dalam *server* basisdata. Jika mampu masuk ke dalam *server*, maka *server* akan menganggap perintah tersebut sebagai SHUTDOWN, mengeksekusinya dan seketika koneksi ke *server* basisdata terputus.



Gambar 15: Ilustrasi arsitektur sistem AMNESIA [Sumber: 14, 15].

2.2 Konsep AMNESIA

AMNESIA merupakan singkatan dari *Analysis and Monitoring for Neutralizing SQL Injection Attacks*, adalah salah satu konsep proteksi yang mampu mendeteksi dan mencegah serangan SQL *injection* pada aplikasi-aplikasi *web* berbasis Java. Konsep ini diperkenalkan oleh seorang mahasiswa dari *Georgia Institute of Technology*, yakni William G.J. Halfond bersama profesornya Alessandro Orso [14, 15].

Konsep proteksi ini menggunakan pendekatan berbasis model, yakni memeriksa kesamaan antara struktur *query* dari hasil analisis statis dengan struktur *query* yang dihasilkan secara dinamis berdasarkan masukan-masukan yang diberikan oleh *user*. Dengan kata lain, ada dua kunci utama yang mendasari konsep ini, yaitu:

a) Informasi mengenai struktur *query* yang dihasilkan oleh suatu aplikasi *web*, diperoleh dari dalam *source-code* milik aplikasi *web* itu sendiri.

b) Serangan SQL *injection*, yakni dengan terinjeksinya pernyataan-pernyataan yang tidak seharusnya ditambahkan ke dalam *query*, tentunya akan merusak struktur *query* yang diperoleh pada langkah pertama.

Oleh karena itu, langkah awal yang dilakukan dalam konsep ini adalah menggunakan sebuah program analisis statis untuk menganalisa *source-code* milik aplikasi *web*. Selanjutnya, secara otomatis membentuk sebuah model dari struktur *query* asli yang dimiliki atau dihasilkan oleh aplikasi *web* tersebut.

Kemudian, pemantauan dilakukan pada langkah selanjutnya (saat *runtime*), yakni memeriksa apakah struktur *query* yang dihasilkan secara dinamis berbeda dengan struktur *query* yang dihasilkan oleh program analisis statis atau tidak. Jika ditemukan perbedaan struktur, maka dianggap bahwa *query* dinamis tersebut bersifat ilegal, dan akan dicegah hak aksesnya ke basisdata, kemudian melaporkannya kepada administrator atau *developer* aplikasi *web* yang bersangkutan.

Konsep AMNESIA terdiri dari empat langkah utama, yaitu: identifikasi *hotspot*, pembentukan model *query* SQL, instrumentasi aplikasi, dan r*untime monitoring*. Sedangkan arsitektur sistem AMNESIA terdiri dari tiga modul utama. Modul-modul tersebut mengimplementasikan keempat langkah utama. Ketiga modul tersebut diantaranya yaitu:

a) Modul Analisis: Modul ini mengimplementasikan langkah pertama dan kedua (identifikasi *hotspot*, dan pembentukan model *query* SQL) menggunakan bantuan pustaka JSA. *Input* untuk modul ini adalah berupa aplikasi *web* berbasis Java, sedangkan *output* yang dihasilkan adalah berupa daftar *hotspot* beserta model-model *query* SQL-nya masing-masing.
b) Modul Instrumentasi: Modul ini mengimplementasikan langkah ketiga (instrumentasi aplikasi) menggunakan bantuan pustaka InsECT (*Instrumentation, Execution, and Collection Tool for Java*), yang telah dikembangkan oleh Chawla dan Alessandro [18]. *Input* untuk modul ini adalah aplikasi *web* berbasis Java beserta daftar *hotspot*-nya, sedangkan *output* yang diperoleh adalah berupa hasil instrumentasi dari setiap *hotspot* tersebut.
c) Modul *Runtime Monitoring*: Modul ini mengimplementasikan langkah keempat (*runtime monitoring*) yang juga menggunakan bantuan pustaka InsECT. *Input*-nya adalah berupa *string query* dinamis beserta ID dari *hotspot* yang bersesuaian dengan *query* tersebut. Kemudian modul ini memeriksa kesamaan antara *string query* dinamis dan model *query* SQL statis.

Gambar 15 menampilkan ilustrasi dari arsitektur sistem AMNESIA. Pada fase statis, modul analisis dan modul instrumentasi mengolah aplikasi *web* sebagai input, dan kemudian menghasilkan dua macam *output*, yaitu: aplikasi *web* yang telah diinstrumentasi, dan model *query* SQL untuk setiap *hotspot* yang terdapat di dalam aplikasi *web* yang bersangkutan. Sedangkan pada fase dinamis, modul *runtime monitoring* memeriksa *query-query* dinamis yang diperoleh dari hasil interaksi *user* dengan aplikasi *web*.

2.3 PHP-SQL Parser

Modul ini dikembangkan oleh Justin Swanhart, seorang *principal consultant* di Percona, Inc. California. PHP-SQL *Parser* merupakan sebuah modul yang berfungsi untuk memilah pernyataan *query* SQL, hingga terurai menjadi elemen-elemen terkecil pembentuk pernyataan *query* tersebut [22]. Hasil penguraian ditampung ke dalam larik atau *array*, sehingga struktur *query* hasil penguraian dapat dilihat dengan jelas dan mudah.

Modul ini ditulis dalam bahasa PHP dan berorientasi pada dialek MySQL. Meski saat ini hanya fokus pada dialek MySQL, namun menurut penulisnya modul ini dapat diperluas menggunakan dialek-dialek SQL lainnya. Prosedur penggunaan modul ini cukup mudah, yakni cukup dengan meletakkan *file* php-sql-parser.php ke dalam direktori yang diinginkan, kemudian menghubungkannya menggunakan fungsi PHP require_once(), include_once(), atau sejenis lainnya. Ada dua cara untuk menggunakan modul ini, diantaranya yaitu:

a) Menggunakan konstruktor:

Gambar 16: Penerapan modul PHP-SQL *parser* menggunakan konstruktor [Sumber: 22].

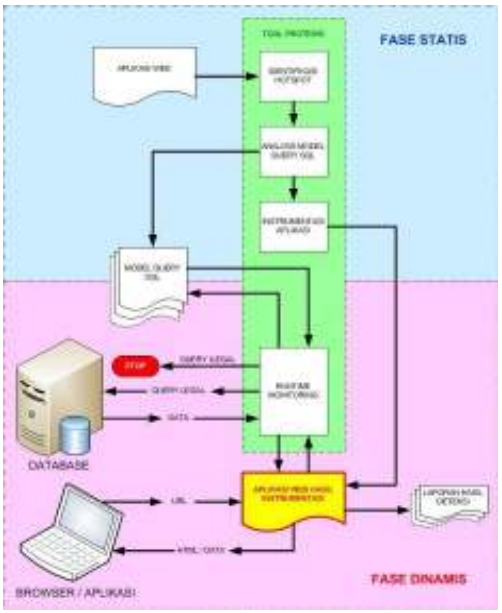
b) Menggunakan *method* parse():

Gambar 17: Penerapan modul PHP-SQL *parser* menggunakan *method* parsed() [Sumber: 22].

3.HASIL PENELITIAN DAN PEMBAHASAN

3.1 Arsitektur Sistem Proteksi Hasil Penelitian

Penelitian ini mengadopsi teknik proteksi AMNESIA dari segi konsepnya saja, sehingga arsitektur sistem proteksi hasil penelitian tidak jauh berbeda dengan konsep aslinya. Gambar 18 menampilkan arsitektur sistem dari hasil penelitian yang telah penulis lakukan.



Gambar 18: Arsitektur sistem proteksi hasil penelitian [Sumber: Hasil Penelitian].

Pada awalnya, yakni pada fase statis, aplikasi *web* dianalisis untuk mencari keberadaan *hotspot* pada setiap *file*-nya. Setelah

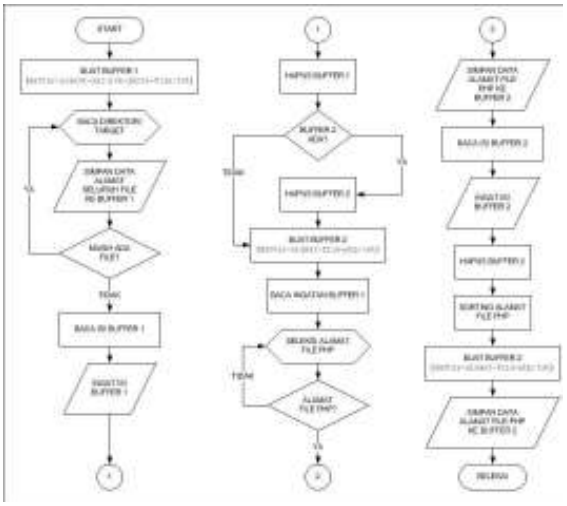
hotspot ditemukan, *string query* SQL diekstrak dari dalam setiap file tersebut, untuk kemudian dibentuk sebuah model *query* SQL untuk masing-masing *hotspot* tersebut. Model-model ini nantinya akan digunakan pada saat *runtime monitoring*, untuk mendeteksi apakah SQLIA ada atau tidak. Langkah akhir dari fase statis adalah instrumentasi aplikasi, yakni melakukan sedikit perubahan pada struktur program milik aplikasi *web* yang mengandung *hotspot*. Instrumentasi ini berjalan otomatis, sehingga cukup efektif dan efisien.

Kemudian pada fase dinamis, aplikasi *web* yang telah diinstrumentasi diakses oleh *user*. Segala permintaan *user* yang mengandung permintaan operasi *database* melalui *hotspot*, akan selalu dipantau melalui proses *runtime monitoring*. Apabila proses *runtime monitoring* mendeteksi adanya SQLIA, maka dengan segera permintaan operasi *database* dihentikan, dan kemudian pesan pendeteksian ditampilkan ke *browser*. Selain menampilkan pesan tersebut, proses *runtime monitoring* juga menyimpan laporan ke dalam sebuah *database* terpisah. Hal ini tentu sangat berguna untuk memantau seberapa banyak SQLIA yang menyerang *website* tersebut, sehingga pemilik *website* dapat mengambil tindakan lebih lanjut.

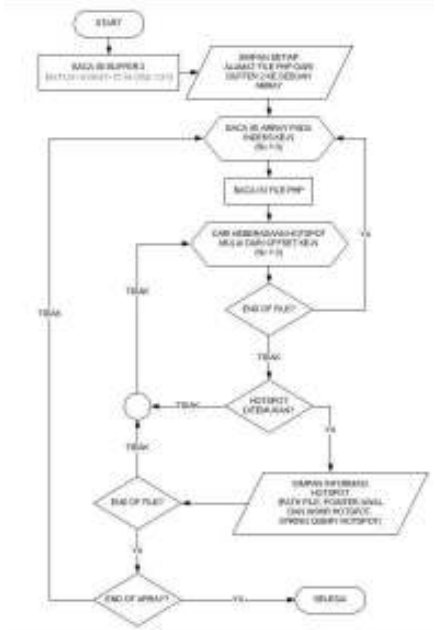
3.2 Identifikasi *Hotspot*

Proses ini menggunakan dua buah modul, yaitu: php-files-crawler.php, dan hotspot-crawler.php. Modul yang pertama, yakni php-files-crawler.php, bertugas untuk mengumpulkan informasi mengenai alamat dari seluruh *file* PHP yang ada di dalam *website* target. Sedangkan modul hotspot-crawler.php, bertugas untuk mengidentifikasi *hotspot* dari setiap *file* PHP, yang sebelumnya telah diketahui alamatnya dengan bantuan modul pertama. Proses identifikasi *hotspot* secara ringkas dijelaskan menggunakan diagram alir yang ditampilkan pada Gambar 19 dan Gambar 20.

Modul php-files-crawler.php memiliki sebuah fungsi yang bersifat rekursif. Hal ini berguna untuk menelusuri seluruh *node* yang ada di dalam direktori target. Berdasarkan Gambar 19, fungsi rekursif mulai bekerja setelah *buffer* 1 dibuat. Perlu diketahui, bahwa sebelum *buffer* 1 dibuat, direktori target harus diinisialisasi terlebih dahulu. Hal ini harus dilakukan, karena akan menentukan keberhasilan langkah-langkah selanjutnya. Kemudian, setelah informasi mengenai seluruh alamat *file* diperoleh, langkah selanjutnya adalah memilah alamat-alamat tersebut. Alamat yang tertuju pada *file* PHP akan disimpan ke dalam *buffer* 2, sedangkan yang lainnya tidak. Daftar alamat *file* PHP selanjutnya perlu di-*sorting*, agar lebih mudah menelusuri letak *hotspot* dari dalam setiap *file* PHP yang ada di dalam daftar tersebut.



Gambar 19: Diagram alir dari modul php-files-crawler.php [Sumber: Hasil Penelitian].



Gambar 20: Diagram alir dari modul hotspot-crawler.php [Sumber: Hasil Penelitian].

Daftar alamat *file* PHP kemudian dibaca dan disimpan ke dalam sebuah *array*, sehingga setiap indeks dalam *array* tersebut memuat sebuah alamat *file* PHP. Selanjutnya *file-file* PHP dari setiap indeks *array* yang bersesuaian diperiksa, apakah *hotspot* ditemukan di dalam *file* tersebut atau tidak. *Hotspot-hotspot* yang ditemukan, kemudian disimpan ke dalam *file-file* yang bersesuaian, dan diberi nama sesuai dengan *pointer* atau ID *hotspot* yang bersangkutan.

3.3 Pembentukan Model *Query* SQL

Langkah selanjutnya adalah membentuk model *query* SQL berdasarkan informasi *hotspot* yang telah diperoleh dari langkah sebelumnya. Untuk melakukan hal ini diperlukan dua modul, yaitu: query-string-extractor.php, dan query-model-builder.php. Model ini sebenarnya adalah berupa sebuah fungsi yang bertugas untuk menyaring berbagai masukan yang diberikan oleh *user*. Setiap variabel yang tersirat di dalam *string query* SQL di setiap *hotspot*, dipilah dan diekstrak, untuk kemudian dijadikan sebagai parameter-parameter masukan bagi setiap fungsi tersebut. Gambar 21 dan Gambar 22 manampilkan diagram alir yang menjelaskan secara ringkas mengenai proses pembentukan model *query* SQL.

Setiap *file hotspot* yang didapat dari langkah sebelumnya dibaca, kemudian *string query* SQL yang ada di dalamnya diekstrak. Selanjutnya hasil ekstraksi disimpan ke dalam sebuah *file* dengan nama yang sesuai dengan ID *hotspot*-nya masing-masing. Kemudian, seluruh variabel yang terdapat di dalam setiap *string query* SQL diekstrak dan dilampirkan ke dalam *file* yang sama.

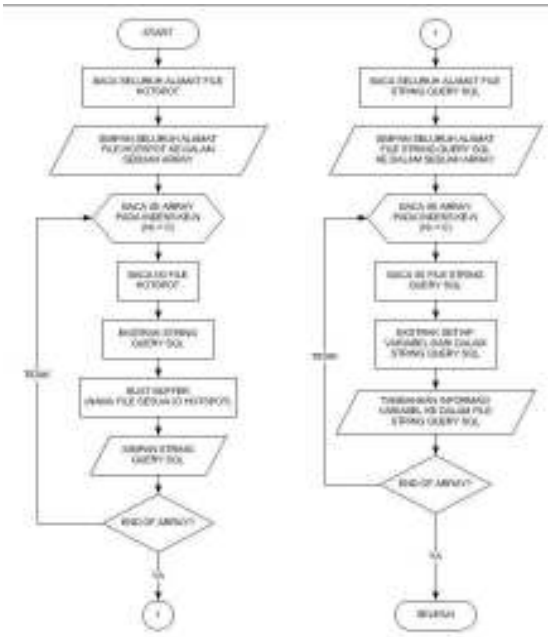
Modul query-model-builder.php membentuk sebuah model *query* SQL untuk setiap *hotspot* yang ada. Model tersebut adalah berupa sebuah fungsi, dimana parameter-parameter fungsi tersebut diperoleh berdasarkan variabel-variabel dari setiap *string query* SQL sebelumnya. Struktur model *query* SQL kurang lebih tampak seperti pada Gambar 23.

Sebagaimana dijelaskan sebelumnya, model *query* SQL sebenarnya adalah berupa sebuah fungsi dari masing-masing *hotspot* yang bersesuaian. Gambar 23, merupakan *source code* dari salah satu model *query* milik *website* yang menjadi media

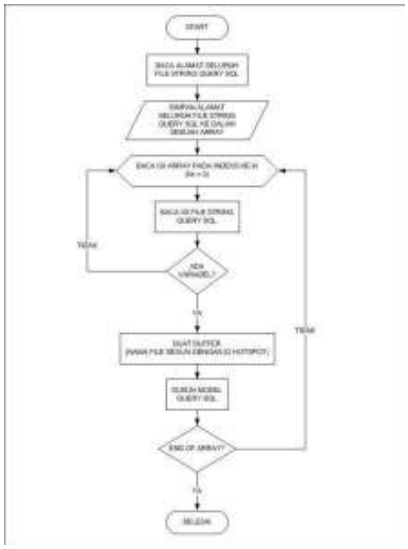
eksperimental penelitian. Seperti terlihat dalam gambar tersebut, nama fungsi diawali dengan "A3S_", dan diikuti dengan ID *hotspot* yang bersangkutan.

Untuk mendeteksi keberadaan SQLIA, maka model *query* SQL dibantu oleh sebuah kelas PHP-SQL *parser*. Sebagaimana dijelaskan dalam Bab II, kelas ini berfungsi untuk menguraikan *string query* SQL, hingga terpisah menjadi elemen-elemen dasar SQL pembentuknya. Penguraian ini disimpan ke dalam sebuah *array*, yang merupakan representasi dari pohon elemen-elemen dasar SQL pembentuk *string query* SQL tersebut. Oleh karena itu, hasil ini dimanfaatkan untuk mendeteksi kehadiran SQLIA, yang mana merupakan serangkaian elemen-elemen dasar SQL, yang disusun sedemikian rupa untuk dapat berkomunikasi dengan *server database*.

Setiap parameter fungsi, akan diperiksa menggunakan bantuan kelas PHP-SQL *parser*. Apabila SQLIA terdeteksi, maka fungsi dengan segera melaporkan kondisi tersebut. Dalam hal ini, laporan disampaikan dengan dua cara, yakni: tampilan pesan peringatan di *browser*, dan data penyerangan SQL *injection* beserta data eksekusi *hotspot* yang disimpan ke dalam *database* terpisah.



Gambar 21: Diagram alir dari modul query-string-extractor.php [Sumber: Hasil Penelitian].



Gambar 22: Diagram alir dari modul `query-model-builder.php` [Sumber: Hasil Penelitian].

Gambar 23: Ilustrasi struktur model *query* SQL [Sumber: Hasil Penelitian].

3.4 Instrumentasi Aplikasi

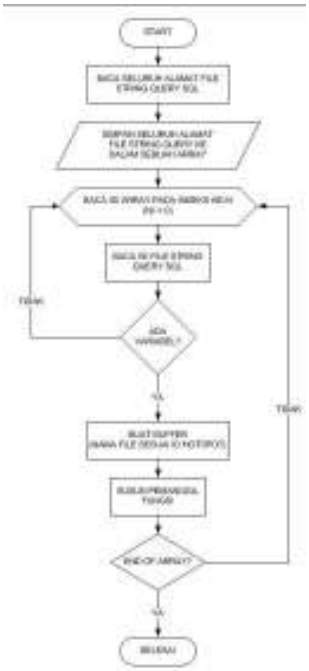
Model *query* SQL yang telah terbentuk harus dipanggil dari dalam *file* target milik *website*. Oleh karena itu, langkah instrumentasi dilakukan dengan cara mengganti setiap *hotspot* dengan pemanggil fungsi atau *monitor* atau model *query* SQL yang bersesuaian. Instrumentasi aplikasi memerlukan dua modul, yaitu: monitor-caller-builder.php, dan instrumentator.php.

Penjelasan ringkas mengenai proses dari kedua modul tersebut dapat dilihat pada Gambar 24 dan Gambar 25.

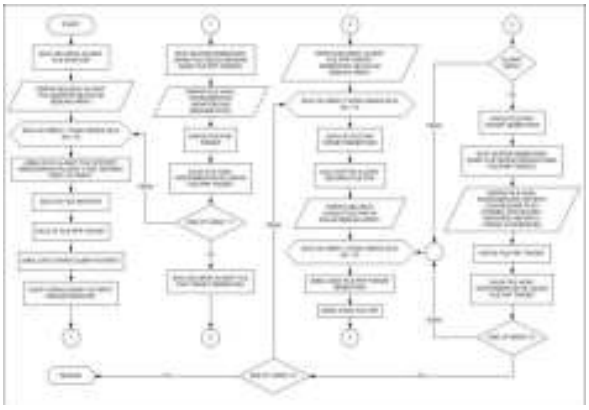
Gambar 24 memang identik dengan Gambar 22, yakni sama-sama menggunakan file *string query* SQL sebagai sumber data. Perbedaan terletak pada *buffer* yang digunakan. Hasil operasi dari modul monitor-caller-builder.php adalah berupa *file* teks, yang berisi pemanggil fungsi atau *monitor*. Sedangkan modul query-model-builder.php menghasilkan *file* PHP, yang berisi deklarasi fungsi. Selain itu, hasil kedua operasi tersebut diletakkan ke dalam direktori yang terpisah.

Pada awalnya, modul instrumentator.php membaca alamat dari seluruh *file monitor* yang telah terbentuk dari langkah sebelumnya. Selanjutnya, alamat-alamat tersebut disimpan ke dalam sebuah *array*, dengan maksud untuk membentuk sebuah *looping* pencarian. Banyaknya *looping* tergantung dari jumlah *hotspot* yang ditemukan, semakin banyak *hotspot*, maka semakin banyak *looping*.

Pembacaan dimulai dari *file monitor* pertama, diikuti dengan pembacaan *file* PHP target, yakni tempat dimana *hotspot* yang bersesuaian dengan *monitor* berada. Kemudian, setiap *hotspot* yang bersesuaian diganti dengan *monitor* atau pemanggil fungsi. Selain itu, *string require path* yang bersesuaian juga ditambahkan. Selanjutnya, hasil instrumentasi tersebut disimpan ke dalam sebuah *file* yang namanya bersesuaian dengan nama *file* PHP target. Lalu *file* PHP target dihapus, dan diganti dengan *file* hasil instrumentasi. Proses ini terus berlangsung hingga seluruh alamat *file monitor* selesai dibaca, yang dengan kata lain menandakan akhir dari *looping array* yang pertama.



Gambar 24: Diagram alir dari modul monitor-caller-builder.php [Sumber: Hasil Penelitian].



Gambar 25: Diagram alir dari modul instrumentator.php [Sumber: Hasil Penelitian].

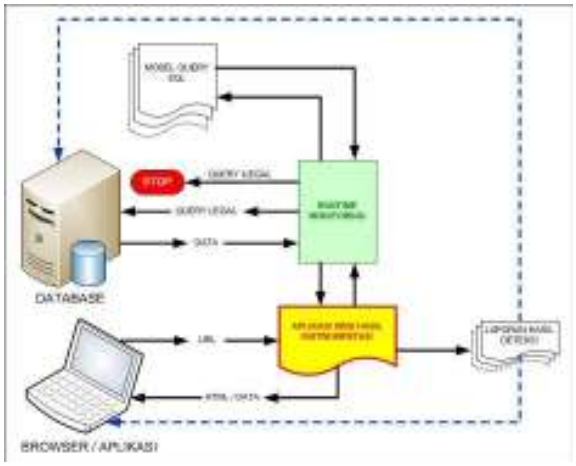
Setelah proses instrumentasi tahap awal selesai, selanjutnya adalah melakukan instrumentasi tahap kedua, yakni penyertaan *string require path* untuk kelas PHP-SQL *parser*, Ip2country, Browser, Reports, dan String Conversion. Selain itu, penyertaan juga dilakukan terhadap variabel-variabel yang diperlukan oleh modul laporan proteksi. Untuk melakukan hal ini, awalnya seluruh alamat *file* hasil instrumentasi tahap awal dibaca, dan disimpan ke dalam sebuah *array* (*Array* 2). Selain itu, diperlukan pula pembacaan terhadap seluruh alamat *file* PHP, yang juga disimpan ke dalam sebuah *array* (*Array* 3).

Pembacaan kedua jenis alamat tersebut, bertujuan untuk menemukan alamat *file* PHP target, yang memuat *hotspot-hotspot* yang telah terdeteksi sebelumnya. Sebelum langkah instrumentasi dilanjutkan, isi *file* hasil instrumentasi tahap awal dibaca (memori sementara). Selanjutnya, apabila file PHP target ditemukan, maka file hasil instrumentasi tahap awal dihapus. Meski dihapus, file tersebut selanjutnya diciptakan kembali, namun kali ini sudah berisi penyertaan *string require path* untuk kelas PHP-SQL *parser*, Ip2country, Browser, Reports, dan String Conversion, serta variabel-variabel yang diperlukan oleh modul laporan proteksi.

Langkah akhir dari tahap instrumentasi adalah menghapus *file* PHP target. Kemudian, *file* hasil instrumentasi tahap kedua, disalin ke lokasi alamat *file* PHP target yang tadi dihapus. Proses ini terus berlangsung hingga seluruh alamat *file* hasil instrumentasi tahap awal selesai dibaca, yang dengan kata lain menandakan akhir dari *looping array* yang kedua dan ketiga.

3.5 *Runtime Monitoring*

Langkah akhir dari sistem proteksi yang dibuat dalam penelitian ini adalah *runtime monitoring*. Proses ini sebenarnya bukanlah berupa modul seperti pada langkah-langkah sebelumnya, namun berupa proses pemantauan yang dilakukan mulai dari aktifnya *monitor*, yakni ketika hotspot yang bersesuaian dengannya diakses, hingga terbentuknya laporan hasil proteksi terhadap SQLIA. Proses *runtime monitoring* dijelaskan secara ringkas dalam Gambar 26.



Gambar 26: Proses *runtime monitoring* yang terjadi pada fase dinamis [Sumber: Hasil Penelitian].

Tabel 4.1: Hasil pengujian *tool* proteksi menggunakan Netsparker [Sumber: Hasil Penelitian].

	Deteksi Jumlah Eksekusi *Hotspot*		Deteksi Jumlah SQLIA		Jumlah Total Serangan	Persentase Keberhasilan *Tool* Proteksi
	Normal	Abnormal	Normal	Abnormal		
Dengan Penerapan Mode UNIQUE	209	0	110	0	319	100%
Tanpa Penerapan Mode UNIQUE	4558	0	363	0	4921	100%

Pada awalnya, *user* mengakses halaman *web* dari aplikasi *web* yang telah diinstrumentasi. *Runtime monitoring* memantau segala aktivitas yang meminta akses ke *database*, khususnya permintaan-permintaan yang mengandung variabel *hotspot*. Apabila terdeteksi adanya permintaan akses *database* yang memicu aktifnya *monitor*, maka dengan segera *model query* SQL dipanggil untuk menganalisa permintaan tersebut. Jika permintaan tidak mengandung SQLIA, maka proses *runtime monitoring* mengijinkannya untuk masuk ke *database*, namun apabila sebaliknya, maka *runtime monitoring* akan menghentikan permintaan akses dan akan membangkitkan laporan hasil pendeteksian. Laporan pertama akan disimpan ke dalam *database* terpisah, sedangkan laporan kedua akan ditampilkan langsung ke *browser* yang digunakan oleh *user*.

3.6 Hasil Uji-Coba *Tool* Proteksi Terhadap SQLIA

Tool proteksi diuji menggunakan perangkat lunak yang dikenal dengan nama Netsparker. Perangkat lunak ini tidak mendefinisikan dengan jelas seperti apa pola serangan SQL *injection* yang dilakukan, sehingga penulis tidak memiliki dokumentasi resmi dan pasti mengenai setiap *string* atau pola serangan yang sebenarnya dilancarkan oleh perangkat lunak tersebut. Untuk mengatasi keterbatasan ini, penulis telah memanfaatkan kemampuan dari *tool* proteksi dalam menghasilkan laporan mengenai segala aktivitas yang terkait dengan eksekusi *hotspot* maupun SQLIA yang dilakukan oleh perangkat penguji tersebut.

Pada awalnya, jumlah serangan dideteksi dengan mengatur struktur dari *database* yang menampung *field-field* laporan, atau lebih tepatnya, melakukan pengaturan terhadap *field* pola_serangan dan pola_eksekusi, yakni dengan atau tanpa mode UNIQUE. Hal ini mengakibatkan munculnya dua macam laporan, yakni laporan dengan kemungkinan adanya pola-pola yang sama, dan laporan dengan pola-pola yang mutlak berbeda-beda. Tabel 1 menampilkan hasil dari pengujian yang dilakukan terhadap *tool* proteksi menggunakan Netsparker.

Deteksi jumlah eksekusi *hotspot* dan jumlah SQLIA masing-masing dibagi ke dalam dua kategori, yaitu: normal dan abnormal. Untuk deteksi eksekusi *hotspot*, kategori normal berisi nilai-nilai variabel *injectable* yang tidak mengandung unsur-unsur SQLIA. Sedangkan kategori abnormal, berisi nilai-nilai variabel *injectable* yang mengandung unsur SQLIA. Sebaliknya, pada deteksi SQLIA, kategori normal berisi nilai-nilai variabel *injectable* yang mengandung unsur SQLIA, sedangkan kategori abnormal berisi nilai-nilai variabel *injectable* yang tidak mengandung unsur SQLIA. Dengan demikian, persentase keberhasilan *tool* proteksi ditentukan dengan rumus sebagai berikut:

$$\%\ Keberhasilan\ = \frac{SQLIA_N - (SQLIA_A + Exec_A)}{SQLIA_N + SQLIA_A + Exec_A} \times 100\% \quad (1)$$

dimana:

$SQLIA_N$:deteksi jumlah serangan SQL *injection* kondisi normal

$SQLIA_A$:deteksi jumlah serangan SQL *injection* kondisi abnormal (*false-positive*)

$Exec_A$:deteksi jumlah eksekusi *hotspot* kondisi abnormal (*false-negative*)

4. KESIMPULAN DAN SARAN

4.1 Kesimpulan

Berdasarkan hasil penelitian yang telah diperoleh, dapat diambil beberapa kesimpulan sesuai dengan perumusan masalah dan tujuan utama penelitian. Beberapa kesimpulan tersebut di antaranya adalah sebagai berikut:

a) Meski penelitian ini masih menemui kendala dalam hal konversi modul JSA dan InsECT, namun sudah dapat dikatakan bahwa konsep AMNESIA dapat diterapkan ke dalam konteks bahasa pemprograman PHP. Hal ini disebabkan karena pada intinya, konsep AMNESIA terdiri dari empat langkah utama, yakni: identifikasi *hotspot*, pembentukan model *query* SQL, instrumentasi aplikasi, dan *runtime monitoring*. Modul JSA dan InsECT hanyalah wujud realisasi dari keempat langkah tersebut menurut pemikiran penciptanya. Dalam penelitian ini, penulis memiliki pemikiran lain untuk merealisasikan keempat langkah tersebut, sebagaimana telah dijelaskan dalam laporan penelitian ini.
b) Sesuai dengan hasil penelitian, terutama pada hasil pengujian yang diperoleh dari serangkaian uji-coba terhadap *tool* proteksi, dapat disimpulkan bahwa *tool* proteksi hasil penelitian telah mampu membuktikan keberhasilannya dalam mendeteksi dan mencegah SQLIA. Namun, hasil ini diperoleh melalui uji-coba yang hanya menggunakan satu macam perangkat penguji saja, yang artinya, persentase keberhasilan belum tentu 100% untuk perangkat penguji lainnya, atau perkembangan teknik-teknik SQLIA baru lainnya.

4.2 Saran

Penulis yakin bahwa hasil penelitian ini masih jauh dari sempurna. Oleh karena itu, ada beberapa hal yang penulis rasa perlu untuk disampaikan demi pencapaian yang lebih baik bagi perkembangan hasil penelitian ini. Beberapa hal tersebut adalah berupa saran-saran, yang sebaiknya direalisasikan agar memperoleh hasil penelitian yang lebih baik. Beberapa saran tersebut, di antaranya adalah sebagai berikut:

a) Penerapan tata bahasa bebas konteks, sepertinya adalah hal yang sangat penting untuk keberhasilan yang lebih baik bagi konsep AMNESIA. Oleh karena itu, konversi modul JSA dan InsECT ke dalam konteks bahasa pemprograman PHP sebaiknya diteliti lebih lanjut.
b) *Tool* proteksi hasil penelitian ini sebaiknya diuji secara *real-time*, atau menggunakan perangkat penguji lainnya. Sehingga diharapkan persentase keberhasilannya dapat terlihat lebih jelas dan akurat.
c) Fitur laporan yang ditampilkan ke *browser* sebaiknya di non-aktifkan, karena ada beberapa informasi penting yang sebaiknya disembunyikan, seperti: *file* target, dan ID *hotspot*. Kedua informasi ini tentunya dapat dimanfaatkan oleh penyerang dalam melancarkan aksinya.
d) Direktori *tool* proteksi sebaiknya tidak diletakkan bersama dengan direktori-direktori yang dapat diakses melalui *browser*. Hal ini diperlukan agar penyerang tidak dapat mengakses isi direktori dari *tool* proteksi secara langsung.
e) Data laporan pendeteksian seharusnya disimpan ke dalam *database* yang terpisah dari *database website*, dan dengan pengaturan yang berbeda pula, atau akan lebih baik lagi apabila data-data tersebut tidak disimpan ke dalam *server database*, melainkan ke dalam *file* terenkripsi. Cara ini diperlukan untuk meminimalisir serangan yang mungkin terjadi terhadap *database* yang terintegrasi dengan *tool* proteksi.
f) Demi keamanan, sebaiknya fase statis dilakukan dalam kondisi *offline*. Selain itu, *file-file* yang dihasilkan dari fase statis yang tidak diperlukan oleh proses *runtime monitoring*, sebaiknya tidak perlu di-*upload*.

DAFTAR PUSTAKA

[1] Meike, Michael, Johannes Sametinger, and Andreas Wiesauer (2009). "*Security in Open Source Web Content Management Systems*," IEEE Security & Privacy. 7.44–51.
[2] Anonymous, *OWASP Top Ten Project*, https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, diakses tanggal 18 April 2012.
[3] Tajpour, A. and M.J.Z. Shooshtari 2010, 'Evaluation of SQL Injection Detection and Prevention Techniques', *2nd International Conference on Computational Intelligence, Communication Systems and Networks, Liverpool, United Kingdom, 28-30 July 2010*, IEEE Computer Society, USA, 216-221.
[4] Halder, Raju and Agostino Cortesi 2010, 'Obfuscation-Based Analysis of SQL Injection Attacks', *IEEE Symposium on Computers and Communications, 22-25 June 2010, Riccione, Italy*, IEEE Computer Society, USA, 931-938.
[5] Balasundaram, I. and E. Ramaraj Dr., Prof. (2011). "*X – Log Authentication Technique To Prevent Sql Injection Attacks*," International Journal of Information Technology and Knowledge Management. 4. 323-328.
[6] Roy, Sangita *et al*. (2011). "*Detecting and Defeating SQL Injection Attacks*," International Journal of Information and Electronics Engineering. 1. 38-46.
[7] Alserhani, Faeiz *et al*. 2011, 'Event-based alert correlation system to detect SQLI activities', *AINA '11 Proceedings of the 2011 IEEE International Conference on Advanced Information Networking and Applications*, IEEE Computer Society, Washington, 175-182.
[8] Pratap, V.K. *et al*. (2011). "*Font Level Tainting: Another Approach For Preventing SQL Injection Attacks*," International Journal of Computer Science & Technology, 2, 52-56.
[9] Jiao, M. *et al*. 2011, 'High-Interaction Honeypot System for SQL Injection Analysis', *International Conference of Information Technology, Computer Engineering and Management Sciences, Nanjing, China, 24-25 September 2011*, IEEE Computer Society, USA, 274-277.
[10] Rawat, Romil *et al*. (2011). "*Safe Guard Anomalies against SQL Injection Attacks*," International Journal of Computer Applications. 22. 11-14.
[11] Kai-Xiang, Z. *et al*. 2011, 'TransSQL: A Translation and Validation-Based Solution for SQL-injection Attacks', *First International Conference on Robot, Vision and Signal Processing, Kaohsiung, Taiwan, 21-23 November 2011*, IEEE Computer Society, USA, 248 - 251.
[12] Telang, Sonal *et al*. 2012, 'Development of an Effective Runtime Defense Algorithm for Web Application Security', *IACSIT Coimbatore Conferences, International Proceedings of Computer Science and Information Technology*, IACSIT Press, Singapore, 6-10.
[13] Rawat, Romil and S.K. Shrivastav (2012). "*SQL injection attack Detection using SVM*," International Journal of Computer Applications, 42, 1-4.
[14] Halfond, W.G.J. and Alessandro Orso 2005, 'AMNESIA: Analysis and Monitoring for NEutralizing SQLInjection Attacks', *20th IEEE/ACM International Conference on Automated Software Engineering, Long Beach, California, USA, 7-11 November 2005*, IEEE Computer Society, USA, 174-183.
[15] Halfond, W.G.J. and Alessandro Orso 2005, 'Preventing SQL Injection Attacks Using AMNESIA', *28th International Conference on Software Engineering, 20–28 May 2006, Shanghai, China*, IEEE Computer Society, USA, 795-798.
[16] Anonymous, *Programming Languages*, http://wappalyzer.com/categories/programming-languages, diakses tanggal 28 April 2012.
[17] Christensen, A.S., Anders Moller, and Michael I. Schwartzbach, 'Precise Analysis of String Expressions', *Proceedings of the 10th International Static Analysis Symposium, June 2003*, Springer-Verlag, 1–18.
[18] Chawla, Anil and Alessandro Orso, 'A Generic Instrumentation Framework for Collecting Dynamic Information', *Online Proceeding of the ISSTA Workshop on Empirical Research in Software Testing, Boston, MA, July 2004*, USA.
[19] Clarke, Justin (2009). *SQL Injection Attacks and Defense*. Burlington: Syngress Publishing, Inc.
[20] Halfond, W.G.J., Jeremy Viegas, and Alessandro Orso, 'A Classification of SQL-Injection Attacksand Countermeasures', *Proceedings of the IEEE International Symposium on Secure Software Engineering, Arlington, VA, March 2006*, USA.
[21] Anonymous, *Web Applications: What are they? What of them?*, http://www.acunetix.com/websitesecurity/web-applications.htm, diakses tanggal 2 Mei 2012.
[22] Anonymous, *PHP SQL Parser*, http://code.google.com/p/php- sql-parser/, diakses tanggal 10 Juni 2012.
[23] Minamide, Yasuhiko and Nobuo Otoi, *PHP String Analyzer*, http://www.score.is.tsukuba.ac.jp/~minamide/phpsa/, diakses tanggal 10 Juni 2012.
[24] Kosuga, Yuji *et al.* 2007, 'Sania: Syntactic and Semantic Analysis for Automated Testing against SQL Injection', *Twenty-Third Annual Computer Security Applications Conference, 10-14 December 2007, Miami Beach, Florida*, IEEE Computer Society, Los Alamitos, California, 107-117.
[25] http://www.mavitunasecurity.com, diakses tanggal 17 Juni 2012