Especialización de GBA para seguridad en la web

Bontempi, María Paula
Bormape, Sabrina

March 6, 2008

Resumen General

- Área: seguridad de aplicaciones web
- Técnica: análisis estático
- Estructura del trabajo:
  - describir el problema de seguridad y privacidad

Resumen General

- Área: seguridad de aplicaciones web
- Técnica: análisis estático
- Estructura del trabajo:
  - describir el problema de seguridad y privacidad
  - presentar vulnerabilidades de seguridad (especialmente las de inyección)

Resumen General

- Área: seguridad de aplicaciones web
- Técnica: análisis estático
- Estructura del trabajo:
  - describir el problema de seguridad y privacidad
  - presentar vulnerabilidades de seguridad (especialmente las de inyección)
  - explicar Análisis de Expresiones String Basado en Gramáticas, una técnica basada en Sistemas de Tipos

Resumen General

- Área: seguridad de aplicaciones web
- Técnica: análisis estático
- Estructura del trabajo:
  - describir el problema de seguridad y privacidad
  - presentar vulnerabilidades de seguridad (especialmente las de inyección)
  - explicar Análisis de Expresiones String Basado en Gramáticas, una técnica basada en Sistemas de Tipos
  - mostrar las adaptaciones necesarias para detectar vulnerabilidades de inyección

Resumen General

- Área: seguridad de aplicaciones web
- Técnica: análisis estático
- Estructura del trabajo:
  - describir el problema de seguridad y privacidad
  - presentar vulnerabilidades de seguridad (especialmente las de inyección)
  - explicar Análisis de Expresiones String Basado en Gramáticas, una técnica basada en Sistemas de Tipos
  - mostrar las adaptaciones necesarias para detectar vulnerabilidades de inyección
  - describir algunos detalles de la implementación del prototipo

Proyecto Marco

- Proyecto: "Plataforma híbrida de seguridad para protección de aplicaciones web"
- Colaboración entre
  - LIFIA
  - la sección Corelabs de la empresa Core Security Technologies
- Desarrollo desde marzo de 2006 hasta junio de 2007
- Tema: estudiar técnicas de análisis estático y dinámico para detección de vulnerabilidades

Vulnerabilidades en aplicaciones web

Descripción del Problema

- Crecimiento de las aplicaciones web
  - estándar para brindar servicios online (foros de discusión, ventas, actividades bancarias y gubernamentales, etc.)
- Facilidad de acceso
  - beneficio para los usuarios del servicio
  - riesgo de ataques incrementado
- Vulnerabilidades en el código de la aplicación
  - en particular, vulnerabilidades de inyección

Descripción del Problema

- Propósito de este Trabajo de Grado
  - estudiar la utilización de una técnica de análisis estático para detectar vulnerabilidades de seguridad de tipo inyección
  - Análisis de Expresiones String Basado en Gramáticas (GBA, por su nombre en inglés)

Vulnerabilidades

- ¿Qué es una vulnerabilidad en una aplicación web?
  - Es un defecto de programación
    - permite a un atacante alterar el comportamiento de la aplicación en beneficio propio
  - Es una imperfección en el código
    - usuarios mal intencionados pueden usarla para revelar información confidencial, realizar operaciones privilegiadas, etc.

Vulnerabilidades de Inyección

- Vulnerabilidades de inyección
  - suceden cuando la aplicación
    - arma un string con código en otro lenguaje
    - usando datos dados por el usuario
    - y luego ejecuta tal programa
  - la entrada contiene contenido sintáctico
  - permiten que la semántica de la expresión armada no sea la esperada por el diseñador

Vulnerabilidades de Inyección

- Tipos de inyección
  - SQL injection
  - cross-site scripting
  - shell injection

Vulnerabilidades SQL injection

- Ejemplo:
  - Código PHP para autenticar un usuario

```
get($email); get($pincode)
$query = "SELECT *
          FROM users
          WHERE email='" . $email . "'
          AND pincode=" . $pincode;
$result = mysql_query($query);
```

  - Si el atacante provee en ($email)

    "alice@host' OR '0'='1"

  - ¡Se produce un ataque!

(4)

Vulnerabilidades SQL injection

- Ejemplo (cont.):
  - Notar el uso de las comillas simples "desbalanceadas"
  - El atacante puede introducir operadores lógicos donde sólo se esperaba un email
  - La consulta producida será:

```
"SELECT *
 FROM users
 WHERE email='alice@host' OR '0'='1'
 AND pincode="
```

  - El resultado es independiente del código de pin provisto

Vulnerabilidades SQL injection

- Ejemplo (cont.):
  - Otra posibilidad es que provea cualquier cosa en la dirección de e-mail y en `$pincode`

    `"0 OR 1=1"`
  - La consulta producida será:

    ```
    "SELECT *
     FROM users
     WHERE email='cualquierCosa'
     AND pincode=0 OR 1=1"
    ```
  - El atacante se autentica sin importar que nombre suministre

Soluciones a las vulnerabilidades de inyección

- Validar la entrada para evitar contenido sintáctico
- Sin embargo, esto no es trivial y es propenso a errores
  - Los caracteres clasificados como contenido sintáctico dependen del contexto en el cual se usa la expresión
    - comillas simples y espacios en SQL injection
    - comentarios, puntos y comas, etc. en shell injection
  - El contexto también depende de cómo se usa la entrada
    - e.g. constantes strings vs. códigos de pin numéricos
  - Los chequeos son específicos de cada base de datos; puede aparecer una vulnerabilidad por cambiarla
- Por lo tanto se necesita inspección de código

Soluciones a las vulnerabilidades de inyección

- La inspección de código depende del momento en el que se requiera:
  - Si aún no se ha realizado la codificación:
    estudiar técnicas de construcción de programas que garanticen la ausencia de estas vulnerabilidades
  - En caso contrario:
    realizar un análisis del código posterior a la codificación para detectar dichas vulnerabilidades y eliminarlas
- Nos concentraremos en la segunda solución dado que la empresa Core requiere un análisis posterior a la codificación por el tipo de tarea que pretende automatizar (auditoría de código)

Análisis de Strings

Sistemas de Tipos Estáticos

- ¿Qué son los sistemas de tipos estáticos?
  - Herramientas que obtienen información de un programa estáticamente para determinar propiedades de los mismos antes de ejecución
  - Asocian información a cada parte de un programa
  - Se basan en el texto del programa, teniendo en cuenta la semántica
- Principal motivación para su uso
  - Cada programa con un tipo calculado estáticamente está libre de algunas clases de errores durante la ejecución.
  - Garantiza cierta corrección del código y ayuda a obtener un programa más eficiente.

Sistema Hindley-Milner

- Sistema de tipos estático adoptado como base por la mayoría de los lenguajes funcionales.
- Características generales
  - Tipos básicos
  - Tipos Compuestos. Se destacan las funciones $(A \longrightarrow B)$
  - Polimorfismo paramétrico
    - Permite que a una expresión que puede ser tipada de infinitas maneras, se le asigne un tipo que sea más general que todos ellos. Por ej: id :: $a \longrightarrow a$
  - Algoritmo de inferencia automático sin requerir ninguna información de tipo en el programa

Sistema Hindley-Milner con restricciones

- Diversas extensiones usan restricciones (constraints)
  - Sistemas para tipar registros
  - Overloading
  - Subtyping
- Todas estas extensiones desarrollan
  - Teorías de constraints similares
  - Algoritmos de inferencia similares

Sistema Hindley-Milner con restricciones

- Sistema HM(X)
  - Framework general para sistemas estilo Hindley-Milner con constraints
  - Diversos sistemas pueden ser obtenidos instanciando el parámetro X a un sistema de constraints específico.
    - X sistema de constraints trivial $\Rightarrow$ sistema Hindley-Milner
    - X álgebra de subtipos $\Rightarrow$ sistema Hindley-Milner con subtipos

Sistema Hindley-Milner con restricciones

- El análisis en HM(X) se divide en dos partes:
  - Inferencia de tipos
    - Calcula un conjunto de restricciones
    - Es genérico para todos los X
    - Similar a dar una especificación del problema
  - Resolución de constraints
    - Calcula una solución para todos los contraints
    - El algoritmo de resolución depende de qué constraints se usan
    - Similar a implementar una solución que satisfaga la especificación

GBA

- GBA es una instancia de HM(X) para el análisis de strings
- Motivación: uso de strings en contextos inadecuados
- ¿Cuál es el problema?
  - Mantenimiento de programas cuyos strings deben respetar un formato externo fijo
- Propósito de GBA: aliviar el mantenimiento y debugging
- Permite comprobar que los valores de una expresión string son elementos de un lenguaje libre de contexto dado por alguna gramática de referencia
- Lenguaje utilizado: Lambda cálculo aplicado con funciones recursivas, un condicional que chequea si un string está vacío y operaciones para construcciones de strings

GBA

- Enfoque
  - Se enriquece el tipo String con variables de lenguaje ($Str(\varphi)$)
  - El sistema de constraints predica sobre dichas variables

$$
\begin{array}{rcll}
C & ::= & true & \text{–siempre verdadera} \\
 & & \mid\ \tau \stackrel{.}{\leq} \tau' & \text{–relación de subtipado} \\
 & & \mid\ r \stackrel{.}{\subseteq} \varphi & \text{–inclusión de lenguaje} \\
 & & \mid\ C \wedge C & \text{–conjunción}
\end{array}
$$

donde r es una secuencia de símbolos terminales y variables de lenguaje:

$$
r \quad ::= \quad \varphi \mid a \mid \epsilon \mid r \cdot r
$$

GBA

- Enfoque (cont.)
  - Las operaciones sobre strings generan constraints de inclusión de lenguaje

$$
\begin{array}{lll}
\omega & : & \forall\varphi.(\omega \mathrel{\dot{\subseteq}} \varphi) \Rightarrow \mathit{Str}(\varphi) \\
{+}{+} & : & \forall\varphi_1\varphi_2\varphi.(\varphi_1{+}{+}\varphi_2 \mathrel{\dot{\subseteq}} \varphi) \\
 & & \quad \Rightarrow \mathit{Str}(\varphi_1) \rightarrow \mathit{Str}(\varphi_2) \rightarrow \mathit{Str}(\varphi) \\
\text{if} & : & \forall\alpha,\varphi.\mathit{Str}(\varphi) \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha
\end{array}
$$

  - El análisis de expresiones strings se realiza en dos partes, pues es instancia de HM(X)

GBA

- Inferencia de tipos con constraints
  - Se infiere el tipo del programa con las restricciones que satisfacen los strings producidos por el mismo
- Resolución de constraints
  - Se resuelven las restricciones a partir de una gramática de referencia G
  - La gramática de referencia provee la estructura del lenguaje (HTML, SQL, etc.)
  - Se determina cuáles de las asignaciones de los no-terminales de G a variables de lenguaje satisfacen las restricciones
  - Estos no-terminales aproximan la forma de los strings producidos

GBA

- Tomando estas dos partes juntas podemos preguntar:
  - "¿El valor de una expresión de tipo string es derivable de un no-terminal dado en la gramática de referencia?"
- Su respuesta permite validar que la salida de un programa respeta ciertos formatos.
  - Por ejemplo HTML

GBA

- Ejemplo (en Haskell)

```
build_list :: String -> String -> String
build_list x y =  "<p>" ++ x ++ "</p>" ++
                   "<ol>" ++
                    "<li>" ++ y ++ "</li>" ++
                   "</ol>"

my_program = build_list "<b>Productos</b>" "Producto 1"
```

  - ¿El programa produce código HTML válido?
  - El tipo en Haskell no provee esa información

GBA

- Asumiendo que `build_list` tiene el siguiente tipo GBA

$$C \quad ::= \quad \texttt{<p>++}\varphi_1\texttt{++</p>
++}\varphi_2\texttt{++
} \mathrel{\dot{\subseteq}} \varphi$$

$$\vdash \texttt{build_list} :: \forall \varphi \varphi_1 \varphi_2 . (C) \Rightarrow Str(\varphi_1) \rightarrow Str(\varphi_2) \rightarrow Str(\varphi)$$

- luego `my_program` tiene el tipo

$$\begin{aligned} C_1 \quad ::= \quad & \texttt{<p>++Productos++</p>
++} \\ & \texttt{
++Producto 1++
++
} \mathrel{\dot{\subseteq}} \varphi \end{aligned}$$

$$\vdash \texttt{my_program} :: \forall \varphi . (C_1) \Rightarrow Str(\varphi)$$

GBA

- Pero, ¿`my_program` produce código HTML válido?
  - La resolución de constraints es usada para dar una respuesta
    - Cada variable de lenguaje es asignada a un no-terminal (usando una asignación Θ)
    - Las inclusiones de lenguaje son resueltas usando un proceso parsing-like ($\Theta(\varphi) \Rightarrow^* \Theta(r)$ para cada $r \dot{\subseteq} \varphi$)
    - El resultado de la resolución es una disyunción de todas las asignaciones válidas ($\Theta_1 \vee \Theta_2 \vee ... \vee \Theta_n$)
  - Se usa una gramática para HTML, G_{HTML}, como gramática de referencia
    - Si todas las variables pueden ser asignadas con algún no-terminal de G_{HTML}, luego la respuesta es *sí*

GBA

- Ejemplo (cont.)

```
G_HTML -> <html> H B </html> | H B
H -> <head> <title> P </title> </head>
B -> <body> C </body>
P -> | P P | Productos | Producto 1 ...
C -> | P | C C | <p> C </p> | <b> C </b>
     | <ol> L </ol> | <ul> L </ul> | ...
L -> <li> C </li> | L L
```

  - El proceso de resolución obtiene el tipo resultante

$$\texttt{my_program} : \forall\varphi.[\varphi = \texttt{C}] \Rightarrow \mathit{Str}(\varphi)$$

indicando que el string producido es código HTML válido

Aplicando la Técnica

GBA para seguridad

- Aplicar GBA a la detección de vulnerabilidades de inyección
- Necesitamos poder identificar:
  - variables de lenguaje consideradas sensibles (utilizadas donde una vulnerabilidad podría ser explotada)
  - símbolos no-terminales inseguros (derivan en strings que contienen terminales representando entrada del usuario)
- Estamos interesados en las asignaciones que involucran
  - variables de lenguaje sensibles
  - símbolos no-terminales inseguros
  - (si a una variable sensible se le puede asignar un no-terminal inseguro, hay una vulnerabilidad)

Modificaciones Realizadas

- Duplicamos el alfabeto para diferenciar los símbolos utilizados en el programa de los provistos por el usuario, resultando en
  - una versión pura del alfabeto
  - una versión impura (indicada con una raya sobre cada carácter)
- Agregamos la expresión *prim*
  - Permite introducir funciones primitivas junto con su tipo
  - Similares a variables definidas con *let* pero sin código explícito (e.g. *get*)
- Agregamos un nuevo constraint llamado *sink*
  - Indica que una variable de lenguaje es sensible
  - Permite especificar que la entrada de una función es sensible

Modificaciones Realizadas

- Ampliamos la expresividad de los constraints de inclusión
  - En $r \dot{\subseteq} \varphi$ permitimos que r pueda ser, además,
    - una constante impura ($\overline{a}$)
    - un comodín para caracteres del alfabeto puro (Σ)
    - un comodín para caracteres del alfabeto impuro ($\overline{\Sigma}$)
  - Permite la identificación de *sources*
- Ampliamos la expresividad de la gramáticas
  - Con no-terminales inseguros $G = (N, T, P, S, NI)$
  - Con terminales puros e impuros
  - Con no-terminales reservados SIGMA y D_SIGMA para expresar un caracter válido cualquiera

Modificaciones Realizadas

- En el ejemplo de SQL Injection (6)
  - Sea el tipo de las operaciones básicas:

$$\texttt{get} :: \forall \varphi \varphi'.\epsilon \dot{\subseteq} \varphi', \overline{\Sigma} \cdot \varphi' \dot{\subseteq} \varphi' \Rightarrow Str(\varphi) \rightarrow Str(\varphi')$$
$$\texttt{mysql_query} :: \forall \varphi, \varphi'.sink(\varphi) \Rightarrow Str(\varphi) \rightarrow Str(\varphi')$$

    - El tipo provisto para *get* indica que es un *source*
    - El tipo provisto para *mysql_query* indica que es un *sink*
  - La inferencia da como resultado:

$$\epsilon \dot{\subseteq} \varphi_2, \overline{\Sigma} \cdot \varphi_2 \dot{\subseteq} \varphi_2,$$
$$\epsilon \dot{\subseteq} \varphi_3, \overline{\Sigma} \cdot \varphi_3 \dot{\subseteq} \varphi_3,$$
$$\texttt{"SELECT * FROM users WHERE email = "++}$$
$$\varphi_2\texttt{++ " AND pincode = "++} \ \varphi_3 \dot{\subseteq} \varphi_1,$$
$$sink(\varphi_1) \Rightarrow Str(\varphi_5)$$

Modificaciones Realizadas

- Necesitamos una gramática que exprese la vulnerabilidad:

<SQL>	::=	<HARMLESS>|<DANGEROUS>
<HARMLESS>	::=	SELECT <ROWS> FROM <TABLE> WHERE <HCOND>
<DANGEROUS>	::=	SELECT <ROWS> FROM <TABLE> WHERE <DCOND>
<HCOND>	::=	<HATOM> AND <HCOND>|<HATOM> OR <HCOND>| <HATOM>
<HATOM>	::=	<HSTR> = '<RSYMB>*'|<HSTR> = <RDIGIT>*
<RSYMB>	::=	/* cualquier símbolo excepto $\overline{'}$ */
<RDIGIT>	::=	0|..|9|$\overline{0}$|..|$\overline{9}$
<DCOND>	::=	<DATOM> $\overline{\text{AND}}$ <DCOND>|<DATOM> $\overline{\text{OR}}$ <DCOND> <DATOM> AND <DCOND>|<DATOM> OR <DCOND>| <DATOM>
<DATOM>	::=	<HSTR> = <QUOTE><RSYMB>*<QUOTE>| <HSTR> = <RDIGIT>*
<QUOTE>	::=	'|$\overline{'}$

Modificaciones Realizadas

- La resolución asigna <DANGEROUS> a φ_1
- Para interpretar el resultado, vemos que
  - φ_1 es sensible
  - <DANGEROUS> es un no-terminal inseguro
  - existe una asignación de φ_1 a <DANGEROUS>
  - entonces, la vulnerabilidad fue detectada

Modificaciones Realizadas

- Consideraciones:
  - Nuevas vulnerabilidades pueden expresarse a través de nuevas gramáticas
  - Esto muestra la flexibilidad del enfoque
  - Es necesario un conjunto de gramáticas que capturen diversas vulnerabilidades

Modificaciones a Futuro

- Incorporación de operaciones Head y Tail
- Estudiar la manera de ambiguar gramáticas y destokenizarlas
- Adecuar la técnica para usar gramáticas no ambiguas y basadas en tokens
- Modificar el proceso para detectar el origen de una vulnerabilidad
- Extender el proceso para construir exploits automáticamente
- Abarcar lenguajes con otras características (imperativos, orientados a objetos)

Optimizando la Normalización

- El algoritmo de normalización es costoso
- La regla de mayor influencia es

$$C@(\ C' \wedge \quad r \stackrel{.}{\subseteq} \varphi \wedge \quad r_1 \varphi r_2 \stackrel{.}{\subseteq} \varphi') \quad \Rightarrow \quad r_1 r r_2 \stackrel{.}{\subseteq} \varphi' \ \ \textbf{si} \ \varphi \notin reach(C, \varphi)$$

- GIxTrees
  - Estructura de datos recursiva con estructura no lineal
  - Diseñada para tratar dicha regla
  - Permite capturar transitividades en los contraints de inclusión

GIxTrees

- La definición en Haskell:

```
data GIxNode a = GISym Symbol | GITree (GIxTree a)
data GIxTree a = GIN a [ [GIxNode a] ]
```

- Etapas del tratamiento de un GIxTree:
  - generación
  - pruning
  - interpretación
- Ejemplo:

$\epsilon \dot{\subseteq} \varphi_1$,

φ_2++$\varphi_3 \dot{\subseteq} \varphi_1$,

φ_4++$\varphi_5 \dot{\subseteq} \varphi_2$,

"a" $\dot{\subseteq} \varphi_4$,

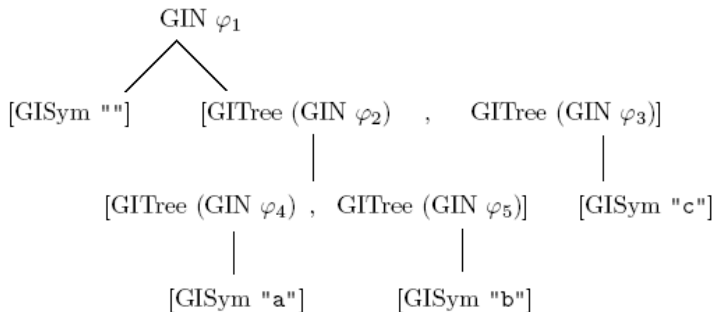
"b" $\dot{\subseteq} \varphi_5$,

"c" $\dot{\subseteq} \varphi_3$,

$\varphi_8 \dot{\subseteq} \varphi_6$

GIxTrees

- El GIxTree formado a partir de la variable φ_1 es:



- La lista de constraints resultante:

$\epsilon \mathrel{\dot{\subseteq}} \varphi_1$, "a"++"b"++"c" $\mathrel{\dot{\subseteq}} \varphi_1$

Conclusiones

- Beneficios al desarrollar un prototipo
- Resultados altamente exitosos –Utilidad de la técnica
- Aporte a la solución del problema de seguridad
- Desafío de realizar investigación aplicada en conjunto con una empresa