Università degli Studi di Verona

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Progetto corso Sicurezza del Software

Offuscatore per il linguaggio Python

Candidati:
Giovanni Bellorio
Matricola VR419955

Kevin Costa
Matricola VR424301

Alessandro Cosma
Matricola VR420117

Anno Accademico 2018-2019

INDICE

1 INTRODUZIONE

Lo scopo di questo progetto è realizzare un offuscatore per il linguaggio Python. Il programma che implementa l'offuscatore sarà realizzato anch'esso in Python e prenderà in considerazione le seguenti tipologie di offuscamento del codice:

- **Layout obfuscation**
- **Data obfuscation**
- **Control Flow obfuscation**

Nel capitolo 2 daremo una panoramica generale sui concetti legati all'offuscamento del codice, spiegando cos'è formalmente un offuscatore, quali sono le sue carateristiche, i suoi pregi e i suoi difetti, infine spiegheremo nel dettaglio le varie tecniche di offuscamento che andremo ad utilizzare.

Nel capitolo 3 verrà presentato il nostro programma per l'offuscamento del codice. Elencheremo le varie scelte progettuali effettuate in fase di progettazione (3.1) e successivamente presenteremo passo passo il codice che implementa l'algoritmo (3.2).

Infine, nel capitolo 4 mostreremo un semplice esempio del suo fuzionamento accompagnato da alcuni test.

2

BACKGROUND

Quando si considera un software sviluppato e distribuito, la maggior parte del suo "know-how" risiede nel prodotto stesso. Risulta quindi essenziale la protezione del suo codice e della sua proprietà intellettuale.

Attacchi di *reverse-engineering* hanno l'obiettivo di attuare analisi statiche e dinamiche del codice, al fine di capire il funzionamento interno di un programma proprietario di interesse. La protezione è necessaria ad ogni livello di funzionamento del programma, a partire dal codice sorgente, fino ad arrivare al codice macchina, passando anche per le informazioni di interesse che il programma da proteggere scambia con l'ambiente di esecuzione.

Esistono varie tecniche di protezione del codice con differenti obiettivi:

- *Misure legali*: leggi, copiright, patenti, brevetti e licenze ideate per proteggere la "forma" e l'idea di un programma.
- *Offuscamento*: rendere difficile il reverse-engineering del programma.
- *Watermarking del software*: inserire una firma o un riconoscimento contro il furto del programma.
- *Tamper-proofing*: rilevare violazioni contro l'integrità del codice o modifiche illegali.

Essendo il nostro progetto mirato alla realizzazione di un offuscatore per codice Python, descriveremo ora in dettaglio il concetto di offuscatore e presenteremo alcune delle principali tecniche utilizzate.

2.1 OFFUSCATORE

Sia:

$$P \xrightarrow{T} P'$$

una trasformazione di un programma sorgente P in un programma P' con $P, P' \in Programs$, cioè l'insieme dei programmi.

$P \xrightarrow{T} P'$ è una trasformazione di offuscamento se P e P' hanno lo stesso *comportamento osservabile*.

Più precisamente, affinchè $P \xrightarrow{T} P'$ sia una trasformazione di offuscamento legale, devono valere le seguenti condizioni:

1. Se P non riesce a terminare o termina in una condizione di errore, allora P' può o non può terminare.
2. Altrimenti, P' deve terminare e produrre lo stesso output di P.

Il *livello di sicurezza* rispetto al reverse-engineering che un offuscatore aggiunge ad un dato programma dipende da vari fattori: raffinatezza della trasformazione di offuscamento, potenza del deoffuscatore, quantità di risorse (tempo e memoria) disponibili al deoffuscatore.

2.1.1 *Impossibilità di offuscare un programma*

Una trasformazione di offuscamento non potrà mai proteggere completamente un programma. Fred Cohen, il primo ad introdurre la diversità dei programmi come difesa contro gli attacchi automatici ai sistemi operativi, affermava che:

"Any protection scheme other than a physical one depends on the operation of a finite state machine, and ultimately, any finite state machine can be examined and modified at will, given enough time and effort. The best we can ever do is delay the attack by increasing the complexity of making desired alterations (security through obscurity)"

[inserire risultati paper "On the impossibility of code obfuscation"]

2.1.2 *Valutare le trasformazioni offuscanti*

Una trasformazione offuscante è valutate in base alle seguenti proprietà:

- **Potenza**: quanto più difficile è comprendere il codice offuscato rispetto a quello originale.
- **Resilienza**: misura quanto una trasformazione regge l'attacco di un deoffuscatore automatico.
- **Costo**: misura quanto overhead computazionale (tempo e spazio) viene aggiunge all'applicazione offuscata.
- **Furtività**: misura quanto bene il codice nuovo introdotto con la trasformazione si integra con il codice originale.

POTENZA Sia T una trasformazione che preserva il comportamento, tale che $P \xrightarrow{T} P'$ trasfroma un programma P in un programma P'. Sia $E(P)$ la complessità di P.

$T_{pot}(P)$, la potenza di T rispetto ad un programma P, è una misura del grado con cui T cambia la complessità di P.

$$T_{pot}(P) = E(P')/E(P) - 1$$

T è una *trasformazione di offuscamento potente* se $T_{pot}(P) > 0$.

RESILIENZA Sia T una trasformazione che preserva il comportamento, tale che $P \xrightarrow{T} P'$ trasfroma un programma P in un programma P'. $T_{res}(P)$ è la *resilienza* di T rispetto ad un programma P.

$T_{res}(P) = one - way$ se l'informazione è rimossa da P tale che P non può essere ricostruito da P'.

$$T_{res} = Resilience(T_{Deobfuscator\_effort}, T_{Programmer\_effort})$$

COSTO Il costo di una trasformazione offuscante è il tempo di esecuzione (o la quantità di spazio occupato) del programma offuscato rispetto a quello originale.

$T_{cost}(P)$, cioè il costo di un programma P, può essere classificato su una scala di quattro punti come segue:

- **caro**: se l'esecuzione di P' richiede esponenzialmente più risorse di P.

- **costoso**: se l'esecuzione di $P^{'}$ richiede $O(n^p), p > 1$ risorse più di P.
- **economico**: se l'esecuzione di $P^{'}$ richiede $O(n)$ risorse più di P.
- **gratuito**: se l'esecuzione di $P^{'}$ richiede $O(1)$ risorse più di P.

FURTIVITA Mentre una trasformazione resiliente potrebbe non essere suscettibile agli attacchi di deoffuscatori automatici, potrebbe comunque essere suscettibile agli attacchi degli umani.

In particolare, se una trasformazione introduce un nuovo codice che differisce ampiamente da ciò che è presente nel programma originale, sarà facile individuare per un reverse engineer. Tali trasformazioni non sono furtive (a.k.a. *unstealthy*).

La furtività è una nozione sensibile al contesto. Il codice può essere nascosto in un programma ma estremamente visibile in un altro.

2.2 TECNICHE DI OFFUSCAMENTO DEL CODICE

In questa sezione andremo a descrivere a livello teorico le tecniche di offuscamento che saranno utilizzate successivamente all'interno del nostro offuscatore.

2.2.1 *Layout Obfuscation*

L'obiettivo del *Layout Obfuscation* è cambiare o rimuove le informazioni utili dal codice senza influire sulle istruzioni reali.

Tecniche principali utilizzate per questo tipo di offuscamento:

- **Rimozione commenti**: vengono rimossi i commenti e i docstring presenti nel codice.
- **Rinominazione degli identificatori** : i nuovi nomi possono utilizzare diversi schemi, numeri, caratteri non stampabili o caratteri invisibili. Inoltre i nomi possono essere riutilizzati, purchè usati in ambiti diversi.

Queste sono trasformazioni *one-way* in quanto la formattazione originale non può essere ripristinata.

Vantaggi:

- potente contro l'attacco umano
- per quanto riguarda i costi è gratuito.
- è molto utile per offuscare *authorship* del codice.

Svantaggi:

- Ha una bassa potenza, in quanto nella formattazione c'è poco contenuto semantico.

2.2.2 *Control Flow Obfuscation*

Il *Control Flow Obfuscation* trasforma un control-flow ben strutturato di un programma eseguibile in una struttura difficile da analizzare da un programma che rileva le trasformazioni del control-flow. Tale offuscamento può ostruire e fermare gli attacchi statici al control-flow del codice eseguibile sotto attacchi di tipo white-box. Questa tecnica costringe l'attaccante ad eseguire attacchi dinamici per ottenere il control-flow, in questo modo diventa molto più difficile da ottenere e aumenta significativamente lo sforzo dell'attacco.
Tecniche principali utilizzate per questo tipo di offuscamento:

- **Predicati opachi**: sono espressioni di valore booleano per le quali il difensore sa se restituiranno true, false o talvolta true e talvolta false
- **CFG Riducibili**: sono grafi che possono essere ridotti a un singolo nodo mediante una sequenza di applicazioni di due regole
- **Predicati opachi dinamici**: una famiglia di predicati opachi correlati, che valutano tutti lo stesso risultato ad ogni esecuzione, ma in diverse esecuzioni possono valutare risultati diversi

Vantaggi:

- potente contro l'analisi statica
- difficili da valutare dall'attaccante per la complessità esponenziale
- è molto utile per offuscare *authorship* del codice.

Svantaggi:

- Vulnerabile all'analisi dinamica

3 OFFUSCATORE PER PYTHON

3.1 SCELTE PROGETTUALI

3.2 ALGORITMO E CODICE

Viene ora presentato il codice che implementa l'offuscatore.

Il codice sorgente è composto da un metodo `main()` che rappresenta l'entry point del nostro programma e da una serie di funzioni che vengono invocate all'interno di tale metodo per svolgere le funzioni di offuscamento.

```
def main():

    print("----------------------------------------")
    print("OBFUSCATION STARTED\n")

    name_src = sys.argv[1] # name of the source file, passed as argument
    name_dest = name_src+"_OBFUSCATED.py" # name of the destination file

    file_SRC = open(name_src, "r")
    file_DEST = open(name_dest, "w")

    # Call 'remove_comments_and_docstrings()' function on the source file object
    # which it is first converted to a string through the read() method.
    # 'out' contains a string with the entire code
    out = remove_comments_and_docstrings(file_SRC.read())

    # Call 'opaque_predicate()' function
    out = opaque_predicate(out)

    # Call 'encoding_literal_data()' function
    out = encoding_literal_data(out)

    # Call 'obfuscate_variables()' function
    out = obfuscate_variables(out)

    # Call 'obfuscate_functions()' function
    out = obfuscate_functions(out)

    # Write the result into file_DEST
    file_DEST.write(out)
    file_DEST.close()
    file_SRC.close()

    print("\nOBFUSCATION ENDED")
    print("----------------------------------------")


if __name__ == '__main__':
    main()
```

All'inizio dell'esecuzione del programma, viene letto il nome del file sorgente, contenente il software Python da offuscare e salvato nella variabile `name_src`: questa informazione deve essere passata come argomento di input all'offuscatore, con una sintassi del tipo:

```
python3 offuscatore.py <nome_file_da_offuscare>.py
```

dove `offuscatore.py` è il nome dell' offuscatore e `<nome_file_da_offuscare>` è il nome del file di input.

Successivamente viene aperto in sola lettura il file sorgente (variabile `file_SRC`) e viene creato un nuovo file, chiamato come il precedente ma con l'aggiunta della suffisso "OBFUSCATED" e salvato nella variabile `file_DEST`: esso rappresenta il file che conterrà l'output del nostro programma cioè il software offuscato.

In seguito vengono chiamate in sequenza le seguenti funzioni:

- **remove_comments_and_docstrings()**: esegue un *layout obfuscation* rimuovendo i commenti e la documentazione;
- **opaque_predicate()**: esegue un *control flow obfuscation*, inserendo predicati opachi;
- **encoding_literal_data()**: esegue un *data obfuscation*, codificando stringhe, array e interi;
- **obfuscate_variables()**: esegue un *layout obfuscation*, rinominando i nomi di variabili;
- **obfuscate_functions()**: esegue un *layout obfuscation*, rinominando i nomi di funzione e il relativo "corpo".

Ognuna di queste funzioni riceve in input il programma (variabile `out`) trasformato dalla funzione di offuscamento precedente, vi applica un certo insieme di trasformazioni offuscanti e successivamente restituisce un output trasformato, che viene poi dato in input alla funzione di offuscamento successiva.

Alla fine del metodo `main()`, viene scritto nel file di output il programma offuscato e vengono chiusi i file di input e output precedentemente aperti.

3.2.1 *Codice di Layout Obfuscation*

remove_comments_and_docstrings()

La prima funzione di layout obfuscation è *remove_comments_and_docstrings*. Questa funzione ha l'obiettivo di eliminare dal codice sorgente tutti i commenti e tutte le stringhe relative alla documentazione del codice presenti al suo interno.

```
def remove_comments_and_docstrings(source):
    """
    Returns 'source' minus comments and docstrings.
    """

    print("-> removing comments and docstrings")

    io_obj = StringIO(source)
    out = ""
    prev_toktype = tokenize.INDENT
    last_lineno = -1
    last_col = 0
    myline = ""
    is_junk = False
```

```
    for tok in tokenize.generate_tokens(io_obj.readline):
        token_type = tok[0]
        token_string = tok[1]
        start_line, start_col = tok[2]
        end_line, end_col = tok[3]
        ltext = tok[4]
        # The following two conditionals preserve indentation.
        # This is necessary because we're not using tokenize.untokenize()
        # (because it spits out code with copious amounts of oddly-placed
        # whitespace).
        if start_line > last_lineno:
            last_col = 0
        if start_col > last_col:
            myline += (" " * (start_col - last_col))
        # Remove comments:
        if token_type == tokenize.COMMENT:
            is_junk = True
            pass
        # This series of conditionals removes docstrings:
        elif token_type == tokenize.STRING:
            if prev_toktype != tokenize.INDENT:
        # This is likely a docstring; double-check we're not inside an operator:
                if prev_toktype != tokenize.NEWLINE:
                    # Note regarding NEWLINE vs NL: The tokenize module
                    # differentiates between newlines that start a new statement
                    # and newlines inside of operators such as parens, brackes,
                    # and curly braces. Newlines inside of operators are
                    # NEWLINE and newlines that start new code are NL.
                    # Catch whole-module docstrings:
                    if start_col > 0:
                        # Unlabelled indentation means we're inside an operator
                        myline += token_string
                    # Note regarding the INDENT token: The tokenize module
                    # does not label indentation inside of an operator (parens,
                    # brackets, and curly braces) as actual indentation.
                    # For example:
                    # def foo():
                    # "The spaces before this docstring are tokenize.INDENT"
                    # test = [
                    # "The spaces before this string do not get a token"
                    # ]
                    else:
                        is_junk = True
                else:
                    is_junk = True
            else:
                is_junk = True
        else:
            myline += token_string

        if token_type == tokenize.NEWLINE or token_type == tokenize.NL:
            if not (is_junk and re.match(r'^\s*$', myline)):
                out += myline
            myline = ""
            is_junk = False

        prev_toktype = token_type
        last_col = end_col
        last_lineno = end_line
    return out
```

Questa funzione sfrutta il modulo `tokenize` il quale fornisce uno scanner lessicale per codice sorgente Python. Essa preserva l'indentazione del codice, essenziale per il corretto funzionamento di un programma Python, inoltre, ogni qual volta una riga viene "svuotata" di tutti suoi token perchè non vi è presenta nessuna istruzione, quest'ultima viene eliminata dal codice per non lasciare inutili spazi vuoti.

obfuscate_variables()

La seconda funzione che esegue il layout obfuscation è *obfuscate_variables*. La funzione prende in input il codice del programma da offuscare (variabile `out`) e ritorna il medesimo codice con i nomi delle variabili offuscate.

Riportiamo il codice:

```
def obfuscate_variables(out):
    """
        Returns 'source' code with variables offuscated.
    """

    print("-> obfuscating variables")

    # Tree object contains the AST of the code
    tree = ast.parse(out)

    # Extraction of all variables defined
    variables = get_variables(tree)

    # Choose new names
    for variable in variables:
        VARIABLES_DICT[variable] = random_sequence()

    # Renomination of all variable occurrences
    tree = rename_variables(tree, VARIABLES_DICT)

    # Convert the new AST to a valid code
    return astor.to_source(tree)
```

Attraverso il comando `ast.parse(out)`, il codice contenuto nella variabile `out` viene convertito in una struttura ad albero e salvata nella variabile `tree`. Questo albero rappresenta l'*abstract syntax tree* del codice sorgente del programma dato in input.

Successivamente vengono estratte tutte le variabili presenti nel programma, attraverso la funzione `get_variables()` e salvate nella lista `variables`. Ad ognuna di esse viene poi associato un nuovo nome casuale, utilizzando la funzione `random_sequence()`.
L'associazione tra il nome della variabile originale e il nuovo nome viene salvata nel dizionario `VARIABLES_DICT`, definito come variabile globale.

Infine, tramite la funzione `rename_variables()`, viene applicata la ridenominazione delle variabili sulla base delle associazioni precedentemente stabilite.

Viene quindi ritornato il codice sorgente del programma, invocando la funzione `astor.to_source(tree)`, che prende come input l'abstract syntax tree, su cui sono state apportate le precedenti trasformazioni offuscanti e ritorna come output il relativo codice sorgente. Questo metodo, che trasforma un abstract syntax tree nel corrispettivo codice sorgente, appartiene al modulo `astor`, esterno alla libreria Python.

Riportiamo ora il codice delle funzioni `get_variables()`, `random_sequence()` e `rename_variables()`, seguito da una breve descrizione.

La funzione `get_variables()` prende come input un abstract syntax tree e ritorna come output un dizionario, contenente ogni variabile presente nell albero (*key*) e il relativo tipo (*value*).

```
def get_variables(tree):
    """
    :param tree:
    :return: a dictionary contains each variable (key) and its type (value)
    contained in the code (passed as AST 'tree')
    """

    variables = {}

    for node in ast.walk(tree):
        # Saved variables declared
        if isinstance(node, ast.Assign) and isinstance(node.targets[0], ast.Name):
            if node.targets[0].id not in variables:
                    variables[node.targets[0].id] = type(node.value).__name__
        if isinstance(node, ast.Attribute):
            if isinstance(node.value, ast.Name):
                if node.value.id == "self":
                    variables[node.attr] = type(node.attr).__name__

    return variables
```

La funzione esegue una visita dell'albero, svolta tramite il metodo `ast.walk()`. In questo caso vengono presi in considerazione i nodi dell'albero che rappresentano un assegnamento (`ast.Assign`) oppure un attributo (`ast.Attribute`); quest'ultimo caso è necessario in presenza di una definizione di classe, in cui una variabile viene anteceduta dal prefisso `self`.

I nomi di variabili e il loro tipo vengono salvati nel dizionario `variables`, ritornato come output dalla funzione.

La funzione `random_sequence()` genera una sequenza casuale di 23 caratteri, scelti in un alfabeto di soli due elementi: il numero '0' oppure la lettera 'O'. La sequenza creata è sempre diversa dalla precedenti sequenze generate.

```
def random_sequence():

    seq = generate_sequence()

    if seq not in RANDOM_SEQUENCES_SET:
        RANDOM_SEQUENCES_SET.add(seq)
        return seq
    else:
        random_sequence()

def generate_sequence():

    cryptorand = SystemRandom()

    sequence = []

    for i in range(22):
        flip = random.randint(1, 2)
        if flip == 1:
            sequence.append("o")
        else:
            sequence.append("O")

    random.seed()
    cryptorand.shuffle(sequence)

    return "O"+"".join(sequence)
```

Questa funzione utilizza il metodo `generate_sequence()` per generare la sequenza. Quest'ultimo utilizza il modulo `randint` e `SystemRandom` della libreria `random`. Ogni sequenza generata, prima di essere ritornata alla funzione `random_sequence()`, viene mescolata tramite il metodo `cryptorand.shuffle()`: questo ci permette di avere meno probabilità di generare dupliacati.

Successivamente, se la sequenza generata è una sequenza valida (non è un duplicato), viene salvata nell'insieme `RANDOM_SEQUENCES_SET`, definito come globale e ritornata come output dalla funzione; altrimenti viene scartata e ne viene generata una nuova.

La funzione `rename_variables()` prende in input un abstract sintax tree (parametro `tree`) e un dizionario contenente le associazioni tra nomi originali delle variabili e i nuovi nomi generati (parametro `variables`). Ritorna in output l' abstract sintax tree con i nomi sostituiti.

```
def rename_variables(tree, variables):
    # Rename names of variables used (general usage)
    for node in ast.walk(tree):
        if isinstance(node, ast.Name) and node.id in variables:
            node.id = variables[node.id]
        if isinstance(node, ast.Attribute) and node.attr in variables:
            node.attr = variables[node.attr]

    return tree
```

4

ESEMPIO DI FUNZIONAMENTO

BIBLIOGRAFIA