Laborator 9
Instrumente pentru auditarea avansată securității aplicațiilor web

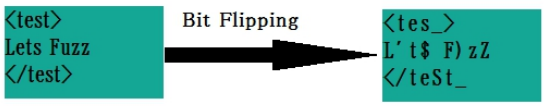
Despre așa zisa securitate a aplicațiilor Web nu mai insist (am discutat destul la curs) o căutare simplă pe Internet vă poate demonstra magnitudinea fenomenului care afectează inclusiv date personale a sute de milioane de utilizatori câteodată.
În domeniul analizei securității aplicațiilor web și nu numai acestor aplicații se utilizează în general o serie de metode cum ar fi analiza statică și dinamică, execuția simbolică și fuzificarea.
Analiza statică se realizează asupra codului și se urmăresc diverse aspecte la nivel gramatical, lexical, anumite entități semantice și evident nu necesită executarea programului (oricum are o rata mare de alarme false ceea ce era de așteptat).
În cazul analizei dinamice aceasta este realizată în timpul execuției (supravegheate a programului) și oricât de avansate instrumente sunt la ora actuala depinde masiv de experiența celor care o realizează.
Execuția simbolică analizează la nivel simbolic intrările programului și după executarea acestuia se aplica constrângerile definite de program pentru a găsi ce subset de intrare cauzează un anumit tip de comportament sau rezultat. Deși suna bine nu este eficientă aplicarea ei în cazul unor aplicații de mari dimensiuni.
Fuzificarea este o tehnică bazată pe furnizarea de intrări invalide aplicației utilizată de obicei pentru a descoperi erorile de implementare. Pentru aplicațiile web este cea mai eficientă abordare având și o bună scalabilitate. Astfel încărcarea suplimentară indusă la nivelul aplicației de aplicarea tehnicii este justificată de rezultate.
Mai jos aveți un tabel sintetic cu privire la analiza comparativă a metodelor anterior menționate.

Metoda	Eficiența	Acuratețe	Scalabilitate
Analiza statică	Destul de mare	Scăzută	În general bună
Analiza dinamică	Scăzută	Mare	Neclară
Execuție simbolică	Scăzută	Mare	Scăzută
Fuzificare	Înaltă	Înaltă	Bună

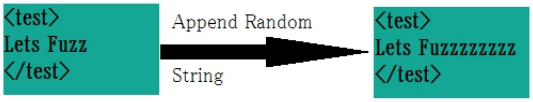
Aplicațiile specifice pot fi clasificate în trei categorii:

- bazate pe mutații,
- bazate pe generații
- evolutive.

Din punct de vedere al aplicării în acest context diferența principală constă în maniera în care generează intrările în program. Astfel primele care sunt cel mai ușor de creat nu țin cont de maniera de organizarea a fluxului de intrare specifică așteptată de aplicație. Ca rezultat nu prezintă un interes deosebit.
Fuzzing-ul bazat pe mutații merge pe idea introducerii unor mici modificări asupra intrărilor existente care mențin valid comportamentul dominant dar pot genera noi comportamente ale aplicației. Metoda poate fi aplicata direct la nivel de parser URL. (https://www.fuzzingbook.org/).
Totuși aceasta intră în categoria așaziselor fuzificatoare tâmpite (dumb fuzzers) deoarece modificările sunt aleatoare sau pseudoaleatoare.
Aveți mai jos un exemplu de utilizare a tehnicii de bit flipping (schimbări ale acestora conform unei funcții sau aleator)



Într-o manieră asemănătoare se poate adauga un sir la sfârșitul unei secvente existente de intrare.



Pot fi variații aleatoare sau de exemplul la nivel de bit. Ca atare s-a ajuns la fuzificarea evolutivă - care este utilizată de asazisele fuzificatoare inteligente. De exemplu Peach un soft destul de utilizat în DevSecOps la ora actuală suporta ultimele două abordări (evident nu este gratuit dar sunt și branch-uri pentru comunitate).

Vectorii pentru fuzificare

Ideal ar fi ca să avem o listă de comenzi sau de combinații de comenzi cu efect de atac și acestea să fie trimise ca atare sau adăugate/combinate cu intrări standard. Mai jos avem câteva exemple simple de astfel de vectori.

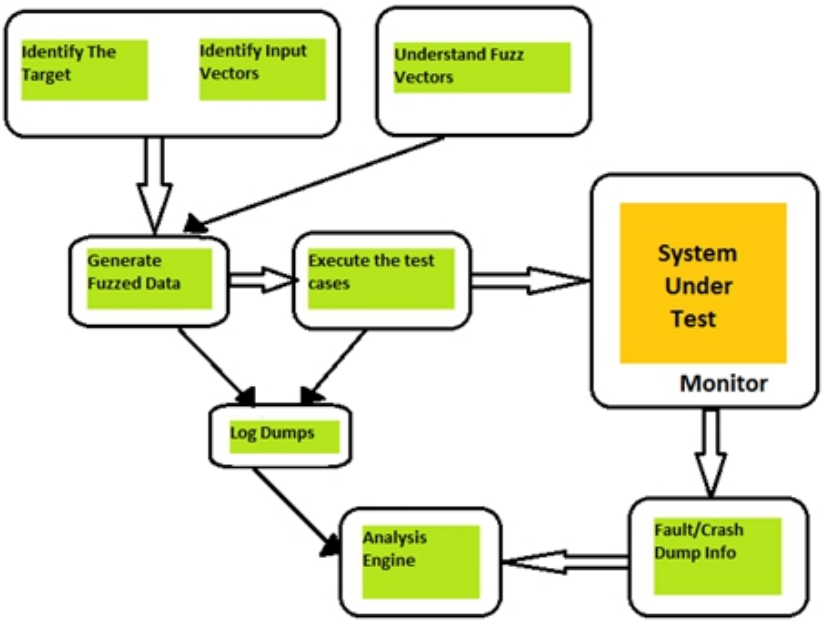
Tipul atacului	Descriere	Vector de atac
Depășirea zonelor rezervate (Buffer Overflow)	Deoarece în acest caz este vorba de limitări în gestiunea unor cantități de date orice este folosit ca intrare poate fi util.	`FIFI * 2` `FIFI * 20FIFI * 200FIFI * 2000`
Erori ale șirurilor de formatare (Format String Errors)	Utilizarea sintaxei specifice limbajului dar în combinații aleatoare poate conduce la erori uneori catastrofale în special atunci când nu s-au luat măsuri de validare ale fluxurilor de intrare	`%s%s%s%s%s%d%s%d%s` `%%200d`

Script-uri cu încrucișări (Cross Site Scripting)	Instrucțiuni suplimentare rău intenționate sunt injectate sub forma de script-uri în codul unor site-uri web nevinovate (de multe ori cunoscute ca fiind curate)	`IMG`
		`SRC=javascript:alert('XSS')> >">script`
		`type="text/javascript"//`
		`alert("XSS")`
		`// &`

Injectarea SQL	Aceasta poate fi pasivă sau activă dar afectează funcționarea corectă a SGBD-ului.	`、 ; drop table user –' or 1=1—`

Utilizarea cadrelor de fuzificare

Indiferent că sunt aplicații gratuite proprietare sau chiar dezvoltate manual (limbajele utilizate pentru dezvoltarea acestora sunt de obicei C, Python sau Ruby) procesul de analiză trebuie să respecte o anumită metodologie care este prezentată sintetic în figura de mai jos.



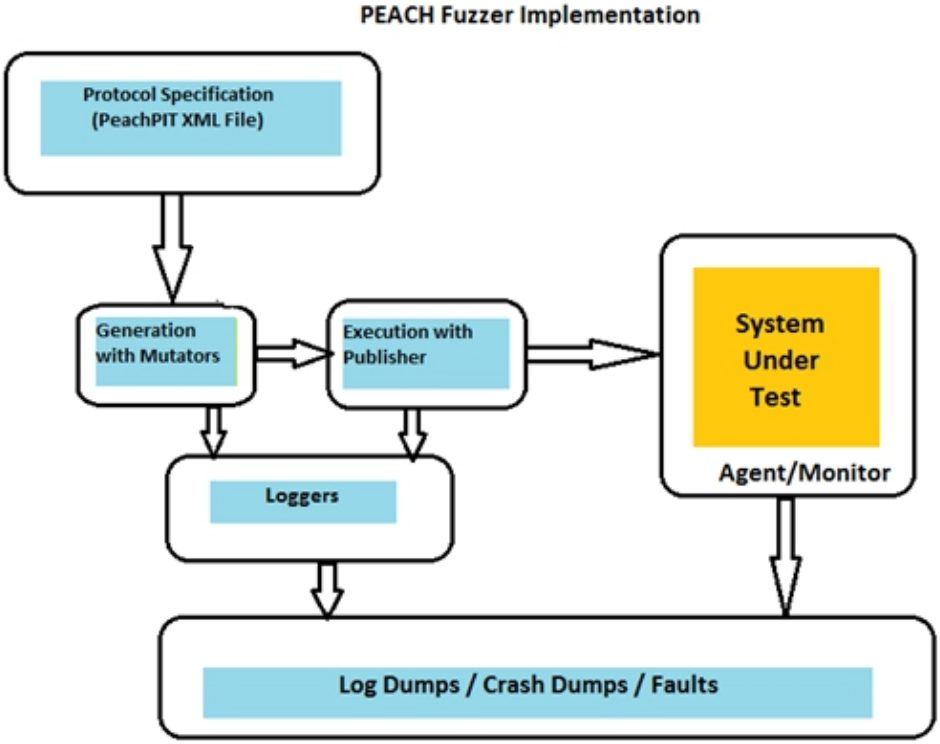
Astfel avem următorii pași

1. Identificarea țintei de testare - având în vedere că nu există aplicații de test care să acopere complet toate variațiile atunci după identificarea țintei se face analiza manierei de implementare și astfel se identifică una sau mai multe ținte primare pentru acea aplicație (de ex serviciu rețea, serviciu web, aplicație web sau chiar aplicații nestandard). Ca atare selecția cadrului de test trebuie făcută ținând cont de aceste obiective.
2. Identificarea vectorilor de intrare - deși prima impresie este că ar fi suficient ca intrările să fie date și argumente în linia de comandă acestea nu sunt suficiente. Astfel putem avea intrări de tip coockie, fișiere de date, variabile de mediu și lista poate continua.
3. Înțelegerea vectorilor de atac (de fuzificare) - maniera de creare a acestora presupune extrem de multe cunoștințe de nivel inferior și superior altfel rămâneți la nivel de copilașilor cu scriptul - de fapt ele sunt 90% din acest atac.
4. Generarea datelor fuzificate - odată identificate secvențele (vectorii) de atac fuzificatorul va genera (exact ca un program normal de test) o serie de variații (suite) de testare pentru a creste posibilitatea unui atac reușit.
5. Trimiterea datelor suitelor de test - depinzând de cadrul de lucru utilizat aceasta rămâne fie la latitudinea utilizatorului fie sunt trimise automat către zonele țintă.
6. Monitorizarea - este extrem de importantă pentru că din analiza reacțiilor sistemului analizat precum și a fluxurilor de date apărute ca urmare a testării se pot trage concluzii cu privire la eficienta atacului precum și informații cu privire la posibile modificări ale vectorilor de atac.
7. Analiza - se realizează asupra informațiilor salvate în cazul procesului de monitorizare.

Începătoșarea cu Piersicuța

peach fuzz se referă la țepii foarte fini (puful) ai frunzelor și fructelor piersicului. Studiile indică că sunt destul de deranjante pentru o serie de dăunători de aici și o serie de denumiri alte softurilor și metodologiei de testare atac. Asta ca să nu faceți confuzie între terminologia asociata logicii fuzzy și cea folosită în laborator :).

PeachPIT XML conține descrierea completă a operațiilor de analiză pe care le va face piersica. Vectorii vor fi generați cu ajutorul a diverși operatori de mutație (mutatori) care permit aplicarea lor atât la nivel de date cat și la nivel de stări. Evident că citirea documentației (vai vai) este esențială altminteri acest exemplu didactic nu va folosi la nimic. Apoi restul procesului de testare este automatizat. Structura modulară a piersicii este prezentata mai jos.



Aici se pot utiliza modele de date care să definească structura unui bloc de date prin specificarea de elemente copil suplimentare (de ex. number și string). Protocoalele mai complexe pot fi împărțite în două părți fiecare având propriul său model de date care poate fi reutilizat iar aceste modele pot fi la rândul lor reîmpărțite în blocuri. Descrierile acestora urmează standard-ul XML. Un exemplu pentru descrierea protocolului http (în cazul Peach) aveți mai jos.

```
<DataModel name="Header">
    <String name="Header" />
    <String value=": " />
    <String name="Value" />
    <String value="\r\n" />
</DataModel>

<DataModel name="HttpRequest">

    <!-- The HTTP request line: GET http://foo.com HTTP/1.0 -->
    <Block name="RequestLine">

        <String name="Method"/>
        <String value=" " type="char"/>
        <String name="RequestUri"/>
        <String value=" "/>
        <String name="HttpVersion"/>
        <String value="\r\n"/>
    </Block>
```

```
    <Block name="HeaderHost" ref="Header">
        <String name="Header" value="Host" isStatic="true"/>
    </Block>

    <Block name="HeaderContentLength" ref="Header">
        <String name="Header" value="Content-Length" isStatic="true"/>
        <String name="Value">
            <Relation type="size" of="Body"/>
        </String>
    </Block>

    <String value="\r\n"/>

    <Blob name="Body" minOccurs="0" maxOccurs="1"/>

</DataModel>
```

Interesant este și modelul de stare care este format dintr-un model propriuzis și cel puțin o stare inițială a cărei atribute specifică prima stare reală din care se pleacă. Cu ajutorul câmpului *action* se efectuează diverse acțiuni în modelul de stare cum ar fi de exemplu trimiterea ieșirii către editor sau citirea unei intrări de la acesta. În modelul de date HttpRequest la început va fi numai o stare și acțiune ca mai apoi să putem adăuga și un copil de tip date care se utilizează pentru a crea și încărca un set implicit de date în model.

```
<Data name="HttpGet">
    <Field name="RequestLine.Method" value="GET" />
    <Field name="RequestLine.RequestUri" value="http://localhost" />
    <Field name="RequestLine.HttpVersion" value="HTTP/1.1" />
    <Field name="HeaderHost.Value" value="http://loclahost" />
    <Field name="Body" value="Test Fuzzzingggg " />
</Data>

<Data name="HttpOptions" ref="HttpGet">
    <Field name="RequestLine.Method" value="OPTIONS" />
    <Field name="RequestLine.RequestUri" value="*" />
    <Field name="HeaderHost.Value" value="" />
</Data>

<StateModel name="State1" initialState="Initial">
    <State name="Initial">
        <Action type="output">
            <DataModel ref="HttpRequest" />
            <Data ref="HttpGet" />
        </Action>
    </State>
</StateModel>

<StateModel name="State2" initialState="Initial">
    <State name="Initial">
        <Action type="output">
            <DataModel ref="HttpRequest" />
            <Data ref="HttpOptions" />
        </Action>
    </State>
</StateModel>
```

După ce sunt completate aceste etape se poate trece la testarea serverului web prin combinarea unui model state cu un publisher (editor).

Tema Instalați o versiune gratuită și încercați să vă jucați, cel indicat este la
`https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce`
Dacă vă doare capul încercați la
https://github.com/MozillaSecurity/peach

Tema pe acasă
Instalați și testați peach fuzz-ul de la gitlab de la adresa
`https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce`
Instalați și testați SecList de la
`https://github.com/danielmiessler/SecLists`

Utilizare AFL++
Este un branch al AFL (american fuzzy lop) mai adaptat vremurilor noastre (cu capul în nori). Întâi să îl instalam. Nu uitați că eu administrarea o fac consola supervisor și lucrez pe bullseye. Cel mai simplu este să testăm o imagine docker care îl conține. În caz ca nu aveți docker instalat.
`apt instal docker* - pentru lenesi`
Pentru a avea o viteză mai mare în cadrul compilării și testării trebuie llvm deci
`apt-get install llvm-dev llvm`
apoi
```
docker pull aflplusplus/aflplusplus
docker run -ti -v /location/of/your/target:/src aflplusplus/aflplusplus
```
Cred că este clar că trebuie să aveți sisteme decente de calcul
Altfel trebuie să îl instalați manual - vezi la pagina lor
În principiu AFL are următorul ciclu de lucru:
1. Se compilează ținta cu compilatoarele AFL pentru a îi adăuga informațiile pentru urmărire continua (un fel de strat de instrumentație mai primitiv).
2. Se creează niște cazuri primare de utilizare de la care va porni procesul de fuzificare.
3. Se va executa AFL peste executabilul anterior generat.
4. Se vor analiza rezultatele.

Pentru un exemplu simplu de utilizare se poate utiliza fuzzgoat care este o aplicație creata intenționat ca să aibă vulnerabilități.
```
git clone https://github.com/fuzzstati0n/fuzzgoat
cd fuzzgoat/
```
După cum am spus aceasta trebuie compilată cu afl++. Comenzile de compilare trebuie date în directorul de baza al fuzzzgoat.
Desi AFL are mai multe compilatoare pentru începători este recomandat *afl-clang-fast*.
Pentru a-l utiliza
`export CC=afl-clang-fast`
În cazul fuzzgoat mergem clasic cu
```
./configure
make
```
Crearea testelor de bază ține din neferice de experienta testorului. Acestea pot fi descrise într-unul sau mai multe fișiere. Și aici este deci valabil GIGO pentru că să nu uităm că acestea sunt simple instrumente de testare care cresc productivitatea și eficienta muncii dar nu pot (încă) gândi în locul angajatului.

Pentru teste este recomandat să li se creeze un director separat de exemplu *afl_in*. Apoi se pot copia niște teste - de exemplu din */bin/ps*.
La fel este bine să se creeze un director separat de ex. *afl_out* pentru fișierele rezultate în urma procesului de testare.
Oricum AFL va crea niște subdirectoare conform cazurilor comune adică

- *crashes* care va reține testele care au condus la căderi catastrofice,
- *queue* - care va reține pe cele care sunt încă în lucru sau vor fi,
- *hangs* - testele care conduc la blocarea aplicației.

În sfârșit putem să îl punem la treabă

```
afl-fuzz -i afl_in -o afl_out -- ./fuzzgoat @@
```

Cred că este clar că trebuie să băgați un pic nasul în documentația mamă - nu este suficient să dați trei comenzi și gata.
Ca observație @@ definește poziția în care AFL++ va insera testele în comanda pentru aplicația testată și nu este obligatorie - se poate utiliza direct stdin.
În interfața de lucru trebuie să fiți atenți la timpii pentru diverse operații și dacă am timpi mari la *last new path* probabil că AFL are probleme în găsirea de căi de execuție deci trebuie să verificați dacă nu cumva testați un executabil care nu a fost compilat cu AFL.
În general lăsați-l să lucreze măcar 50 de cicluri. La sfârșit vă uitați prin ce a raportat în fiecare director.

Tema Instalați și testați AFL++ manual sau din docker - cum vreți - apoi testați măcar amărâtul de fuzzgoat. Ideal ar fi să vă jucați cu un proiect mai complex de cpp realizat de voi dacă nu încercați exemplul de la https://aflplus.plus/docs/tutorials/libxml2_tutorial/