Instrumentarea dinamică a binarelor pentru fuzzing

Scop

Ne dorim să instrumentăm programe binare pentru care nu avem acces la codul sursă. Rolul instrumentării este acela de a oferii informații despre flow-ul programului procesului de fuzzing a.î. fuzzerul folosit să ia decizii mai informate în etapa de modificare a inputului cu scopul de a explora (acoperi) cât mai mult din suprafața codului binar.

Metode posibile

Metodele prin care putem instrumenta dinamic cod binar se pot grupa în două categorii:

1. Folosirea unui emulator
2. Patch-uirea codului binar

Folosirea unui emulator

Fuzzere populare precum afl, aflplusplus, honggfuzz, unicornafl folosesc această metodă.

Acestea să bazează pe modul de funcționare al unui emulator pentru a instrumenta codul la **runtime**.
Atunci când un emulator emulează un binar, acesta face următorii pași:

1. Încarcă binarul
2. Citește un basic block (BB). Un BB este definit [1] ca o listă de instrucțiuni terminate de o instrucțiune de branching.
3. Verifică dacă BB de la această adresă (locație) este cache-uit - a fost deja translatat și acum se află în cache-ul emulatorului
4. Dacă nu este cache, codul BB este translatat din instrucțiunile specifice platformei emulate (guest) într-un set intermediar de instrucțiuni specifice emulatorului; în cazul Qemu, reprezentarea intermediară este pentru modului de translație numit Tiny Code Generator (TCG)
5. TCG translatează instrucțiunile în instrucțiuni specifice platformei gazdă (host)
6. Adaugă instrucțiunile translatate în cache
7. Salvează o mapare de la Program Counter (PC) sursă la PC destinație
8. Execută BB
9. Goto 2

Fuzzerele care folosesc emulatoare se folosesc faptul emulatorul întrerupe și dictează execuția BB cu BB.
AFL++ folosește un patch pentru Qemu prin intermediul căruia instrumentează la runtime fiecare BB și astfel analizează flow-ul programului.

Asemănător funcționează și restul fuzzerelor menționate.

AFL, AFL++ și Hongfuzz folosesc emulatorul QEMU, iar unicornafl folosește emulatorul Unicorn.

Avantaje

Procesul de instrumentare este relativ simplu de implementat.
Se garantează că instrumentarea dinamică nu modifică flow-ul normal al programului.

Dezavantaje

Overheadul adus de emulare este mare și execuția programului poate fi de 2x până la 5x mai lentă.

Patch-uirea codului binar

Această metodă vrea să reducă cât mai mult ovearheadul adus de către emulare.
Principiul este următorul:

1. Analizează instrucțiunile binare
2. Identifică instrucțiunile de interes (BB, apeluri de funcții, jmp-uri, etc)
3. Adaugă instrucțiunile (patch) de instrumentare în jurul zonei de interes
4. Rulează binarul instrumentat. Datorită faptului că acum instrucțiunile sunt executate direct pe CPU, performanța va fi îmbunătățită dramatic

Idea pare simplă și eficientă, dar operația de patching este o problemă foarte dificilă.
Problema principală este că introducerea de noi instrucțiuni în codul binar modifică adresele instrucțiunilor din program. Această offsetare a adreselor aduce probleme pentru instrucțiuni de tipul jmp, goto, etc care folosesc adrese absolute sau relative.
Uneltele care implementează această metodă trebuie să identifice corect toate adresele și să le actualizeze la valorile care țin cont de codul de instrumentare adăugat.
Acesta este un proces dificil și care poate introduce buguri greu de depistat.

Având în vedere faptul că beneficiile de performanță pot fi majore, există mai multe unelte și framework-uri care implementează această metodă: rev.ng, QBDI, afl-dyninst, RetroWrite.

AFL-dyninst este un fork al AFL care abordează problema, dar proiectul nu este menținut activ (ultimul commit a fost în Martie 2018) și deci nu i-am acordat interes.

RetroWrite funcționează doar pentru binare compilate pentru x86_64 și vine cu un set de constrângeri legate de cum trebuie să fost compilat codul (PIC/PIE, cu simboluri, fără excepții C++). Având prea multe constrângeri, nu este generic, așa că nu i-am acordat interes.

QBDI este un framework construit peste LLVM care instrumentează codul la runtime folosind Just In Time (JIT) compilation. Instrucțiunile binare sunt translatate (lifted) în instrucțiuni intermediare mașină (LLVM Machine Intermediate Representation) (a nu se confunda cu LLVM IR). Operația de patching se întâmplă apoi asupra instrucțiunilor MIR și apoi codul este recompilat și executat. Așa cum am menționat anterior, challenge-ul principal este rezolvarea corectă a adreselor în urma inserării noilor instrucțiuni.

Mai multe detalii legate de modul de funcționare sunt disponibile în prezentarea QBDI - YouTube - Implementing an LLVM based Dynamic Binary Instrumentation framework

Rev.ng este un framework care implementează un decompilator.
Acesta pretinde că, dându-i-se un binar ELF, acesta îl poate decompila la nivel de LLVM IR. Odată obținută reprezentarea intermediară LLVM, putem folosi pass-uri de LLVM pentru a instrumenta codul. Datorită faptului că rev.ng translatează codul binar în LLVM IR, modificările aduse prin intermediul pass-urilor de LLVM nu mai suferă de problema offset-ării adreselor. Instrumentarea este adăugată la nivel IR, care apoi este compilat în cod mașină; astfel, compilatorul este cel care se ocupă de generarea corectă a codului binar.
Un alt avantaj pe care îl aduce rev.ng este faptul că putem compila codul IR pentru orice arhitectură suportată de LLVM.

Mai multe detalii despre rev.ng sunt disponibile în blog post-ul: Fuzzing binaries with LLVM's libFuzzer and rev.ng

Rev.ng pare foarte promițător, dar nu am reușit să reproduc pașii din blog post.
Am deschis un issue pe pagina proiectului și aștept un răspuns.

Avantaje

Timpul de rulare este redus semnificativ față de rularea binarului sub un emulator.

Dezavantaje

Procesul de instrumentare este complicat și poate introduce buguri greu de descoperit.
Se poate modifica flow-ul normal al programului din cauza unui bug introdus în timpul patchingului. Detectarea acestei anomalii este foarte dificilă, poate chiar imposibilă pentru cineva care nu cunoaște “flow-ul normal” al programului testat.

Paper descoperit aseară: BinRec: Dynamic Binary Lifting and Recompilation, EuroSys ’20

Metodă propusă

Pentru implementarea unui PoC sunt de părere că folosirea unui emulator este cea mai bună opțiune.
Această metodă este simplă de utilizat (am folosit hongfuzz și afl++ fără probleme) și garantează corectitudinea programului (nu modifică flow-ul), în detrimentul vitezei de execuție a binarului.
Trebuie avut în vedere că targetul nostru este reprezentat de binare pentru dispozitive IoT. Acestea, în mod normal, rulează pe microcontrolere cu frecvențe de MHz, deci este posibil să compensăm overheadul de performanță introdus de emulator prin simplul fapt că emulatorul va rula pe un sistem al cărui procesor rulează cu frecvențe de GHz.