

Tema 4 - Exploit ELFs

- Deadline: 24.01.2021, ora 23:55
- Data de început: 11.01.2021
- Responsabil:
 -  Cristian Vîjeliu

Enunț

Respectand tradiția, Moș Crăciun s-a hotărât să împartă binare copiilor din satul UPB. Deoarece nu poate vedea în viitor, Moșul a scris 2 binare per persoană - unul pentru cazul în care am fost cuminți, iar altul pentru cazul în care am fost pe *naughty list*. Din păcate, curierii mosului au fost ocupați și au reușit abia acum să livreze "cadourile".

1. Analiza binarului - 20p

Chiar înainte de a compila binarele, Grinch s-a strecurat în lăcașul moșului și a adăugat o vulnerabilitate în surse. Deoarece Moșul nu înțelege instrucțiunile assembly și deoarece mai este puțin timp până la Crăciun, acesta vă cere ajutorul.

Codul sursă a fost șters "din greșeală", iar aplicația a rămas în mare parte nedocumentată. De asemenea, etichetele din cadrul programului au fost șterse pentru a evita săpăturile în codul sursă.

La acest task, trebuie să analizați binarul **nice** și să descoperiți la ce adresă se găsește funcția vulnerabilă.

Scrieți în fișierul README adresa funcției vulnerabile și de ce este aceasta vulnerabilă.

2. Spargerea binarului - 40p

Moșul vă mulțumește pentru că l-ați ajutat să identificați zona buclucășă din cod, însă acum dorește să înțeleagă și de ce acea zonă era problematică (tot pentru binarul **nice**). Pentru acest task, trebuie să generați un payload (va fi citit de la stdin) care va face programul să printeze un flag de forma: **NICE_FLAG{<sir_de_caractere>}** (dacă un flag de această formă se regăsește în outputul programului, înseamnă că ați reușit).


3. Spargerea binarului v2 - 40p

Pentru acest task, Moșul are nevoie de voi pentru a găsi vulnerabilitatea, iar mai apoi a sparge binarele copiilor obraznici (în binarul **naughty**).

În mod similar cu subpunctele anterioare, trebuie să identificați vulnerabilitatea și să o documentați în README. De asemenea, trebuie să furnizați un payload care să vă ofere flag-ul, care este sub forma: **NAUGHTY_FLAG{<sir_de_caractere>}** (dacă un flag de această formă se regăsește în outputul programului, înseamnă că ați reușit).

4. Bonus - Shellcode - 20p

Generați un payload astfel încât să obțineți un shell folosind binarul **naughty**. Payloadul va fi consumat de la stdin.


Este suficient ca payloadul să funcționeze pentru cazuri cu  ASLR dezactivat. Pe scurt, ASLR este un mecanism de securitate care asigură că, la fiecare rulare, zonele de memorie ale procesului vor fi plasate la adrese random, greu de ghicit.

Pentru a dezactiva ASLR, puteți rula:

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

Chiar și fără ASLR, adresele de stivă sunt destul de imprevizibile și ar putea diferi de la o mașină la alta, deci este posibil ca payloadul vostru să nu funcționeze direct pe altă mașină. De aceea, important este să documentați în README adresa de stivă pe care o folosiți.

Generați un payload astfel încât să obțineți un shell folosind binarul **naughty**. Payloadul va fi consumat de la stdin.

Este suficient ca payloadul să funcționeze pentru cazuri cu  ASLR dezactivat. Pe scurt, ASLR este un mecanism de securitate care asigură că, la fiecare rulare, zonele de memorie ale procesului vor fi plasate la adrese random, greu de ghicit.

Pentru a dezactiva ASLR, puteți rula:


```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

Chiar și fără ASLR, adresele de stivă sunt destul de imprevizibile și ar putea diferi de la o mașină la alta, deci este posibil ca payloadul vostru să nu funcționeze direct pe altă mașină. De aceea, important este să documentați în README logica din spatele payloadului.

Setup

Pentru dezvoltarea temei puteți folosi mașina virtuală de Linux descrisă în secțiunea **Mașini virtuale** din pagina de resurse.

Trimitere și notare

Pentru descărcarea temei trebuie să intrați .

Introduceți user-ul de cs.curs, iar după veți obține binarele pe care trebuie să le utilizați în rezolvarea temei.

Tema va trebui încărcată pe  vmchecker. Arhiva încărcată va fi o arhivă **zip** care trebuie să conțină:

Precizări suplimentare

- Este **interzisă** atacarea infrastructurii

Resurse ajutătoare

- Tool-uri disponibile care vă pot ajuta: gdb, IDA, radare, objdump, ghidra, pwntools, etc.
- **Laborator 10:** Gestiunea bufferelor. Buffer overflow.

BONUS (another one) - Optimizări folosind AVX (30p)

Search

- Anunțuri
- Bune practici
- Calendar
- Catalog
- Feed RSS
- IOCLA Need to Know
- Regulii și notare
- Resurse utile

Cursuri

- Curs 00: Prezentare
- Curs 01-02: Programe și sistemul de calcul
- Curs 02-03: Arhitectura sistemelor de calcul
- Curs 03: Arhitectura x86
- Curs 04: Reprezentarea datelor în sistemele de calcul
- Curs 05: Reprezentarea datelor în sistemele de calcul - C2
- Curs 06 - 07: Setul de instrucțiuni
- Curs 07: Declararea datelor
- Curs 08 - 09: Moduri de adresare
- Curs 09: Stiva
- Curs 10 - 11: Funcții
- Curs 12: C + asm
- Curs 13: Unelte, utilitare
- Curs 13 - 15: Buffer overflows, ...
- Curs 16 - 17: Optimizări
- Curs 18 - 19: Linking

Laboratoare

- Laborator 01: Reprezentarea numerelor, operații pe biți și lucru cu memoria
- Laborator 02: Operații cu memoria. Introducere în GDB
- Laborator 03: Toolchain
- Laborator 04: Introducere în limbajul de asamblare
- Laborator 05: Rolul registrelor, adresare directă și bazată
- Laborator 06: Lucrul cu stiva
- Laborator 07: Apeluri de funcții
- Laborator 08: Structuri, vectori. Operații pe șiruri
- Laborator 09: Interacțiunea C-assembly
- Laborator 10: Gestiunea bufferelor. Buffer overflow
- Laborator 11: Optimizări
- Laborator 12: Linking
- Laborator facultativ: ARM assembly

Teme

- Tema 1 - printf
- Laborator 07: Apeluri de funcții
- Laborator 08: Structuri, vectori. Operații pe șiruri
- Laborator 09: Interacțiunea C-assembly
- Laborator 10: Gestiunea bufferelor. Buffer overflow
- Laborator 11: Optimizări
- Laborator 12: Linking
- Laborator facultativ: ARM assembly

Teme

- Tema 1 - printf
- Tema 2 - strings
- Tema 3 - AST
- Tema 4 - Exploit ELFs

Table of Contents

- Tema 4 - Exploit ELFs
 - Enunț
 - 1. Analiza binarului - 20p
 - 2. Spargerea binarului - 40p
 - 3. Spargerea binarului v2 - 40p
 - 4. Bonus - Shellcode -

Așa cum veți învăța în anul 3, sistemele de calcul pot fi de 4 tipuri, în funcție de câte instrucțiuni se execută în același timp și câte date se prelucresc în același timp: SISD, SIMD, MISD, MIMD.

- SISD (single instruction, single data) este tipul folosit până acum de voi: o singură instrucțiune se execută în același timp și se lucrează pe o singură dată în același timp.
- MIMD (multiple instructions, multiple data): subiectul mai multor materii din anii următori (APD, SO, ASC, APP). În acest tip de arhitectură, se pot executa mai multe instrucțiuni simultan, care pot opera pe mai multe date simultan, folosind thread-uri (fire de execuție).
- SIMD (single instruction, multiple data): aceeași instrucțiune este executată pe mai multe date simultan. Acest tip este subiectul unei bucati importante din cursul de ASC.
- MISD (multiple instructions, single data): tip foarte rar întâlnit.

După cum probabil vă puteți da seama, procesoarele moderne sunt de tip MIMD, întrucât au mai multe nuclee, care la rândul lor pot avea unul sau mai multe thread-uri. Totuși, aceste procesoare moderne au implementate și instrucțiuni tip SIMD, care sunt scopul acestei părți a temei.

E posibil să fi auzit până acum de SSE și AVX, și să vă întrebați ce sunt. SSE este prima variantă de set de instrucțiuni de tip SIMD implementată pe procesoarele Intel, începând cu Intel Pentium III. AVX (Advanced Vector Extension) este varianta îmbunătățită a instrucțiunilor SSE. Ambele seturi de instrucțiuni sunt folosite pentru operații pe vectori.

Să presupunem că vrem să adunăm 2 vectori a câte 8 elemente fiecare, de tip int. Până acum, s-ar fi folosit o buclă for și s-ar fi adunat element cu element. AVX ne permite să efectuăm toate cele 8 adunări, simultan, folosind doar o funcție, numită întrinească de către dezvoltatori, anume `_mm256_add_epi32`, în cazul nostru.

Funcțiile intrinseci AVX din C sunt denumite în felul următor: `_mm<bit_width>_<name>_<data_type>`, unde:

- `bit_width` este lungimea vectorilor, în biți
- `name` este numele operației (add, sub, etc)
- `data_type` este tipul datelor stocate în vector. În cazul nostru, tipul de date este `epi32`, adică tip întreg, cu semn, pe 32 de biți (int).



Aproape fiecărei funcții AVX din C îi corespunde o instrucțiune în limbajul de asamblare.



Tema are ca scop utilizarea unor funcționalități ale AVX/ AVX2, dar, dacă vreți, puteți rezolva cerințele folosind AVX-512 sau alte seturi de instrucțiuni SIMD mai avansate. Se va acorda punctaj redus dacă sunt folosite doar instrucțiuni SSE. Dacă doriți, puteți folosi instrucțiuni SSE, în combinație cu cele AVX.

Resurse

Scheletul temei se poate găsi pe [git-ul IOCLA](#).

În scheletul temei vi se dau următoarele:

- o funcție scrisă în C, care primește ca argumente 4 vectori, de dimensiune n, cu elemente de tip float, A, B, C, D și realizează calculul $D = A * B * I(n) + \sqrt{C}$; A se consideră vector linie, iar B vector coloană, astfel că $A * B$ va fi un scalar; $I(n)$ este vectorul de dimensiune n care conține numai 1; prin \sqrt{C} se înțelege sqrt aplicat fiecărui element din C
- o funcție scrisă în Assembly x64, care execută operația $C = (A.^2) + 2 * B$, A, B și C fiind vectori cu elemente de tip int, de dimensiune n; prin $(A.^2)$ se înțelege că fiecare element al lui A este ridicat la pătrat

Cerință

Vi se cere să optimizați funcțiile de mai sus, ca timp de execuție, folosind instrucțiunile AVX. Fiind o temă bonus, vi se dă doar tipul de instrucțiuni, nu și ce instrucțiuni, în mod exact, trebuie să folosiți, identificarea lor fiind parte a temei. Puteți începe căutarea cu următoarele link-uri:

- [AVX în C](#)
- [Instrucțiuni AVX](#)



Funcțiile din schelet pot fi scrise intenționat astfel încât să nu fie optime



Pentru a se observa mai ușor efectele optimizării, vă rugăm să treceti în README timpii de execuție, arătați de checker-ul vostru local, și raportul dintre timpii de execuție pentru codul neoptimizat și cel optimizat. Acest lucru nu este obligatoriu.

Testare

Aveți inclus un checker, pentru a verifica dacă implementarea voastră este corectă. Pentru a-l compila, intrați în folder-ul checker/ și executați comanda `SKEL_FOLDER=../skel/ make`, dacă aveți sursele în folder-ul `skel`, sau doar `make`, dacă aveți sursele în folder-ul checker.

Înainte de a executa checker-ul, verificați că procesorul vostru are AVX/ AVX2, cu următoarea comandă: `cat /proc/cpuinfo | grep "AVX"`

Dacă aveți AVX, totul e bine.

Dacă nu aveți AVX pe mașina voastră, lucrurile devin mai complicate: va trebui să executați checker-ul pe [fep.grid.pub.ro](#) și să lucrați acolo la partea aceasta de temă. Pentru a vă ușura munca, vă recomandăm să folosiți [Visual Studio Code](#), cu extensia [Remote SSH](#), pentru a putea edita fișierele sursă direct pe fep.

Pentru a putea face build pe fep, este necesar să faceți niste modificări în Makefile:

- Adaugați la `CFLAGS` `-std=c99`
- Eliminați de la regula checker opțiunea `-no-pie`

Trimitere și notare

Partea de AVX a temei va avea o intrare dedicată pe [vmchecker](#), unde va trebui să încărcați o arhivă care să conțină fișierele sursă, cu optimizările cerute, și fișierul README (altul decât la partea de exploit). Punctarea se va face manual, de către corector.

Punctajul se va împărți astfel:

- 12p - optimizarea funcției scrise în C, folosind AVX
- 12p - optimizarea funcției scrise în Assembly x64, folosind AVX
- 6p - descrierea implementării în README + coding style

Pentru a vă încuraja să implementați o soluție cât mai eficientă, **se vor mai acorda câte 2.5p**, pentru C și Assembly, persoanelor care obțin cel mai bun timp. În acest sens, va exista un clasament în funcție de raportul dintre timpul de execuție al programului neoptimizat și cel al programului optimizat, pe care vom încerca să-l actualizăm cât mai des.

Clasamentul se află aici: [Clasament AVX](#)

Sunt eligibile pentru a primi acest bonus doar temele trimise pana la deadline.






Punctajul maxim se acordă doar dacă **toate** operațiile matematice cerute se fac folosind AVX, acolo unde are sens







Se vor acorda punctaje parțiale.

ioda/tema/tema-4.txt - Last modified: 2021/01/19 19:53 by cristian.vigelle

 Old revisions

 Media Manager  Back to top

 BY-NC-SA  CHIMERIC DE  WS  DOKUMIKI  GET FIREFOX  RSS  XML FEED  WS  DHTML 3.0