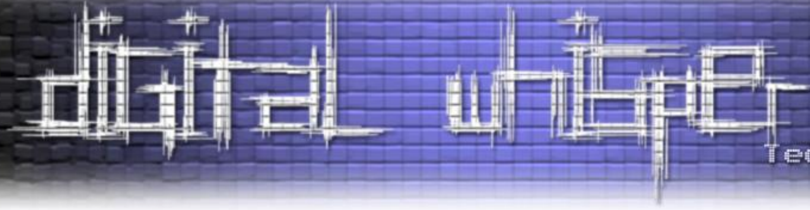


בטוח שאתה זוכר נכון?

מאת ניר עופר / hyprnir

הקדמה

המאמר הבא יעסוק בכתיבת כלי בשפת C שפועל כמו **Cheat Engine** המוכר. מהותו העיקרית של הכלי היא מציאת ערך בזיכרון של תהליך אחר ושינויו לערך רצוי. בעזרת פעולה פשוטה כזו אפשר לעשות הרבה דברים, אך Cheat Engine התפרסם בעיקר בהיותו שימושי במניפולציה על משחקים. המשתמש יכול למשל לחפש את המספר שמייצג את כמות התחמושת שלו בזיכרון של תהליך המשחק ולשנות ערך זה.

שימוש ביכולת זו כולל הרבה ניסוי וטעיה מפני שלא ניתן לדעת איך בדיוק התהליך משתמש בערך הרלוונטי, כך שיכול להיות שאם למשתמש יש 80 פצצות במשחק והוא יחפש את הערך 80 הוא ימצא הופעות רבות שלו בזיכרון התהליך. בעיה נוספת שלא ניתן לפתור באמצעות Cheat Engine היא שמירה של ערכים בצורה שונה ולא גלויה למשתמש. למשל, שמירה של 80 פצצות כ-320. ישנה שיטה עיקרית לצמצום הממצאים, אך גם היא לא מושלמת ואציג אותה בהמשך. אני ממליץ לקרוא את חלקי המאמר יחד עם קוד המקור הרלוונטי.

איך הכל התחיל?

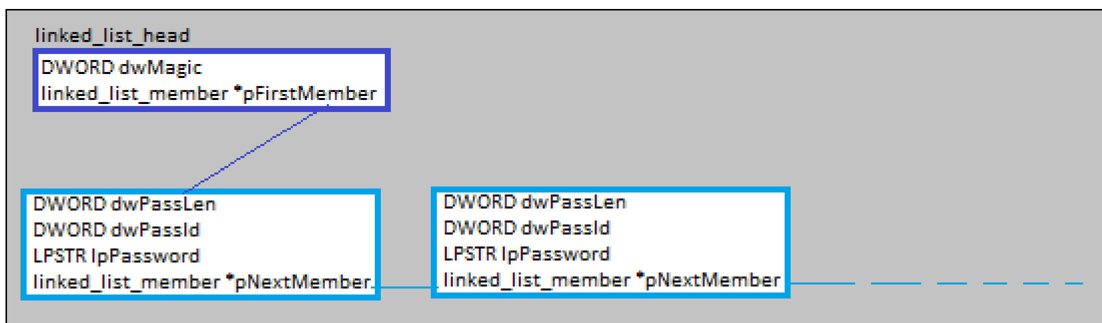
הגעתי לנושא בזכות crackme שכתב דימה פשול. (כתבה שפירסם בגיליון ה-65) ה-crackme מגיע עם הקדמה בצורת קובץ Header כחלק מהקבצים בקישור.

ה-crackme מכיל רשימת סיסמאות עם סיסמאות למערכות שונות. כל סיסמא מיוצגת על ידי struct מסוג linked_list_member ומאפיינים אותה אורך ו-ID. הסיסמא שאנחנו מחפשים היא הסיסמא למערכת הטילים, עם ה-ID 0x30.

הרשימה המקושרת בנויה באופן הבא:

```
#define LIST_HEAD_MAGIC 0x12345678
#define ROCKET_SYSTEM_PASSWORD 0x00000030
#define HR_SYSTEM_PASSWORD 0x00000020
#define ERP_SYSTEM_PASSWORD 0x00000010
#define FACEBOOK_PASSWORD 0x00000005

typedef struct linked_list_member
{
	DWORD                         dwPassLen;
	DWORD                         dwPassId;
	LPSTR                         lpPassword;
	struct linked_list_member    *pNextMember;
}*plinked_list_member;

typedef struct linked_list_head
{
	DWORD                         dwMagic = LIST_HEAD_MAGIC;
	struct linked_list_member    *pFirstMember;
}*plinked_list_head;
```

הרשימה נראית כך:



כאשר מריצים את ה-crackme, כל מה שרואים הוא החלון הבא:

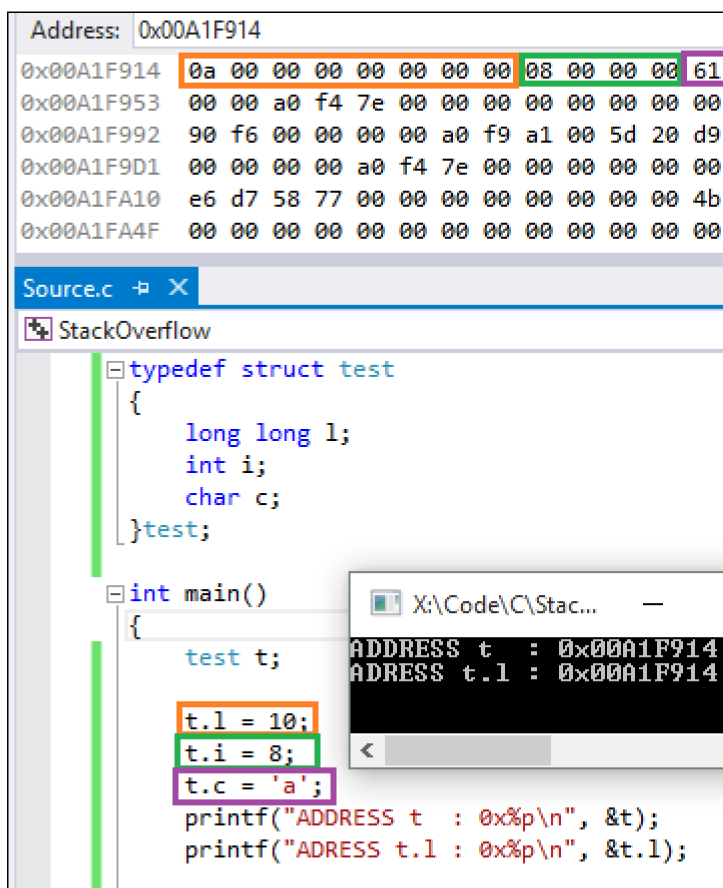
בטוח שאתה זוכר נכון?

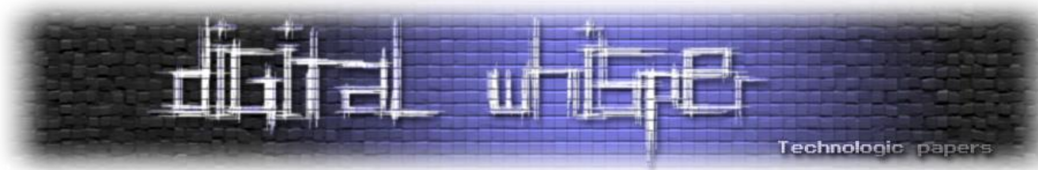
מהמידע שאנחנו מקבלים לגבי ה-crackme מסתמן שהפתרון לא מסובך, אך בעיני הוא מגניב ואלגנטי. מה שעלינו לעשות הוא לכתוב כלי שמגדיר את ה-struct-ים linked_list_head ו-linked_list_member. הכלי מחפש בזיכרון התהליך של ה-crackme את הערך LIST_HEAD_MAGIC המסמן את linked_list_head. משם אפשר להשתמש ב-struct-ים, לרוץ על הרשימה המקושרת ולמצוא את הסיסמא.

כדי לעשות את זה, מומלץ להבין ברמה מסוימת איך struct-ים נראים בזיכרון ולשחק איתם קצת. העיקרון שמשנה לנו הוא הבנה של מהי כתובת הבסיס של struct. (ריפוד בין משתני ה-struct למשל לא משנה לנו במקרה הזה.)

struct-ים הם תאגיד של משתנים שונים שמאוגדים תחת משתנה/מצביע אחד. הם פרוסים ברצף בזיכרון כך שכתובת הבסיס של ה-struct היא כתובת המשתנה הראשון. מיקומם בזיכרון נקבע לפי סדר ההצהרה עליהם. לכן, אם נמצא את הכתובת של dwMagic, נוכל להצביע לכתובת זו עם מצביע של struct מסוג linked_list_head ולעבוד איתו בצורה הגיונית.

המחשה של פריסת struct בזיכרון:





בחרתי לפתור את ה-crackme בעזרת dll injection כי רציתי שהקוד שלי יוכל פשוט לפנות למשתני ה-struct כחלק מהתהליך בלי קריאות זיכרון נוספות עם ReadProcessMemory שהיו דרושות בריצה מחוץ לתהליך. הקריאות הנוספות היו דרושות מכיוון שב-Windows כל תהליך מתנהל במרחב כתובות פרטי ומערכת ההפעלה דואגת להמרה בין כתובת פרטית לכתובת פיזית. ארכיטקטורה זו לא מאפשרת לתהליך פשוט לפנות לזיכרון של תהליך אחר ללא עזרה ממערכת ההפעלה. העזרה הזו באה לידי ביטוי בפונקציות API שונות. בהמשך אציג גם פיתרון ללא dll injection. ההבדלים בין שתי הדרכים זניחים.

פיתרון 1: עם DLL injection

דבר ראשון, יש צורך ביכולת להזריק DLL. אפשר להוריד כלי מוכן, או לכתוב באופן עצמאי. (ניתן לקרוא את המאמר [Code Injection בגיליון 13](http://www.digitalwhisper.co.il) שנכתב על-ידי אוראל ארד).

אני בחרתי לכתוב injector, אך לא אסביר אותו. הוא מצורף למאמר. ועכשיו לכתיבת ה-DLL. נתחיל מהגדרת הקבועים הרלוונטים:

```
#define LIST_HEAD_MAGIC 0x12345678
#define ROCKET_SYSTEM_PASSWORD 0x00000030
```

עכשיו נגדיר את ה-struct-ים. אני שיניתי קצת את השמות כדי שיתאימו לי. חשוב מאוד לשמור על סדר המשתנים כמו ב-struct המקורי. אחרת יוכל לקרות מצב בו נתייחס למשתנה אחד כמשתנה אחר, או שריפוד בין המשתנים יוביל אותנו לקריאה של ערכים לא נכונים:

```
typedef struct LINKED_LIST_MEMBER
{
	DWORD dwPassLen;
	DWORD dwPassId;
	char *lpPassword;
	struct LINKED_LIST_MEMBER *Next;
}LINKED_LIST_MEMBER;

typedef struct LINKED_LIST_HEAD
{
	DWORD dwMagic;
	struct LINKED_LIST_MEMBER *pFirstMember;
}LINKED_LIST_HEAD;
```

תחת DLL_PROCESS_ATTACH בפונקציית DllMain נקרא לפונקציה InjectionMain. בתחילת הפונקציה נגדיר את המשתנים הבאים:

```
MEMORY_BASIC_INFORMATION mbi;
SYSTEM_INFO si;
LPVOID minAddress, maxAddress;
```

מרחב הכתובות הפרטי של תהליך הוא רצף לינארי של כתובות. כדי לסרוק את הזיכרון אנחנו צריכים לדעת מהי הכתובת המינימאלית ומהי הכתובת המקסימאלית. maxAddress תישאר קבועה, וב-minAddress

נשתמש כדי לרוץ על הזיכרון. את הכתובות הדרושות אפשר למצוא באמצעות הפונקציה GetSystemInfo שכותבת את הפלט שלה ל-struct מסוג SYSTEM_INFO. לכן אנחנו מגדירים את si. בין משתני ה-sturct נמצאים המשתנים הבאים:

```
LPVOID lpMinimumApplicationAddress;
```

```
LPVOID lpMaximumApplicationAddress;
```

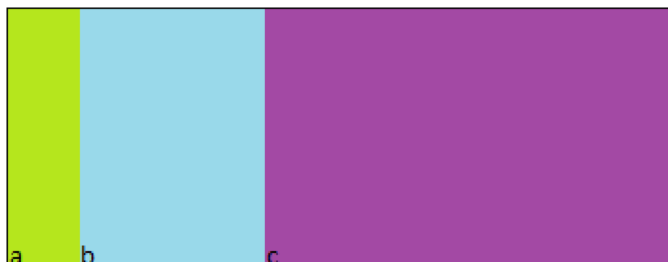
משתנים אלו הם הכתובת המינימאלית והכתובת המקסימאלית שמוקצות למרחב הזיכרון הפרטי של תהליך במערכת. אלה הם הגבולות שבהם נשתמש. MEMORY_BASIC_INFORMATION הוא struct שמוחזר מהפונקציה VirtualQuery ומכיל מאפיינים על בלוק זיכרון. נשתמש בו כדי לאפיין את חלקי הזיכרון השונים שאנחנו סורקים בתהליך. לאחר שהגדרנו את המשתנים נקרא לפונקציה הדרושה ונתחיל לרוץ על הזיכרון:

```
GetSystemInfo(&si);
minAddress = si.lpMinimumApplicationAddress;
maxAddress = si.lpMaximumApplicationAddress;
```

כדי לרוץ על הזיכרון נשתמש בלולאת while ובכל ריצה שלה נגדיל את minAddress. הלולאה תראה כך:

```
while (minAddress < maxAddress)
{
	VirtualQuery(minAddress, &mbi, sizeof(mbi));
	if (mbi.State == MEM_COMMIT) SearchValue(&mbi);
	minAddress = (LPBYTE)mbi.BaseAddress + mbi.RegionSize;
}
```

כדי להבין למה הלולאה עובדת, צריך להבין איך עובדת הפונקציה VirtualQuery. נניח שתחילת הזיכרון נראית כך:



רצף של זיכרון עם אותם מאפיינים מיוצג על ידי אותו צבע. האותיות b,a ו-c מייצגות כתובות הבסיס של רצפי זיכרון עם אותם מאפיינים. בפעם הראשונה שאנחנו קוראים לVirtualQuery, הערך של a הוא minAddress. הפונקציה מתחילה לרוץ על זיכרון התהליך החל מהכתובת שהיא מקבלת וממשיכה לרוץ על הזיכרון עד שהיא נתקלת בכתובת זיכרון עם מאפיינים שונים מהזיכרון שעליו רצה, כלומר, היא הגיעה לסוף של רצף זיכרון עם אותם מאפיינים. במקרה הזה הכתובת היא b. הפונקציה מחזירה struct מסוג MEMORY_BASIC_INFORMATION עם מאפייני רצף הזיכרון. ב-struct זה קיימים המשתנים:

```
PVOID  BaseAddress;
```

```
SIZE_T RegionSize;
```

בטוח שאתה זוכר נכון?

במקרה הזה BaseAddress הוא a, וRegionSize הוא b-a. מידע זה מאפשר לנו לרוץ על כל כתובות התהליך, לפי רצפי זיכרון בעלי מאפיינים זהים. לאחר שאנחנו קוראים לפונקציה אנחנו בודקים:

```
if (mbi.State == MEM_COMMIT && mbi.Protect == PAGE_READWRITE)
```

כלומר, האם לזיכרון הוירטואלי הזה הוקצה זיכרון פיזי. תהליך עובד במרחב כתובות וירטואלי משלו. לזיכרון שבו הוא משתמש בפועל מוקצה זיכרון פיזי במערכת. יתכן שהוא לא משתמש בכל הזיכרון הוירטואלי, ולכן מעניין אותנו רק הזיכרון שלו הוקצה זיכרון פיזי. בנוסף, נבדוק שההרשאות על הזיכרון מאפשרות קריאה וכתיבה. הכתיבה לא קריטית אך אלה ההרשאות על הזיכרון במקרה הזה (אפשר לשחק עם התנאי ולהשמיט את החלק השני. במקרה שלנו אלה המאפיינים של הזיכרון שאנחנו מחפשים). אם הזיכרון אכן עומד בתנאים. נקרא לפונקציה SearchValue.

לאחר מכן נגדיל את minAddress כדי להמשיך את ריצת הלולאה. כדי להגיע לבלוק הבא בזיכרון עלינו לחבר את כתובת הבסיס של הבלוק הנוכחי עם גודל הבלוק. כלומר, נגיד וכתובת הבסיס הנוכחית היא a, וגודל הבלוק הוא b-a, כדי להגיע לכתובת הבסיס הבאה, b, עלינו לבצע את החיבור (b-a) + a. השימוש ב-(LPBYTE) נעשה כדי שהחישוב יהיה מדויק. הגדלה של מצביע תלויה בגודל הערך שעליו הוא מצביע. הוספה של 1 למצביע מסוג *char תוסיף לכתובת שלו 1, אך הוספה של 1 למצביע מסוג *int תוסיף לכתובת שלו 4.

הפונקציה SearchValue מקבלת מצביע מסוג MEMORY_BASIC_INFORMATION המצביע על בלוק זיכרון ובו היא מחפשת ערך לבחירתנו. במקרה הזה הערך הוא LIST_HEAD_MAGIC. מכיוון שהDLL רץ כחלק מהתהליך, הפונקציה פשוט תרוץ על הכתובות ותבדוק את ערכיהן. בפונקציה נגדיר שני משתנים. הכתובת המינימאלית של בלוק הזיכרון והכתובת המקסימאלית. נחפש כל פעם את LIST_HEAD_MAGIC בטווח זה:

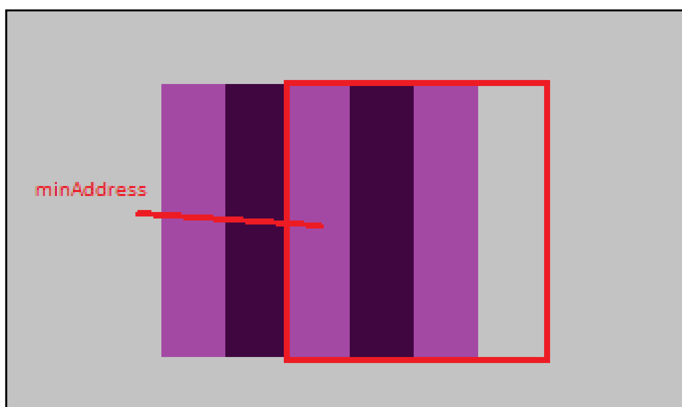
```
LPVOID minAddress, maxAddress;

minAddress = mbi->BaseAddress;
maxAddress = (LPBYTE)minAddress + mbi->RegionSize - sizeof(DWORD);
```

minAddress שווה לכתובת הבסיס של בלוק הזיכרון. maxAddress שווה לסכום כתובת הבסיס וגודל הבלוק. הסיבה לחיסור (sizeof(DWORD מ-maxAddress היא שיש לחסר בית אחד כי אם נשאיר אותו התוצאה היא כתובת הבסיס של הבלוק הבא. מחסירים שלושה בתים נוספים היא כי אנחנו מחפשים ערך DWORD. גודל הערך הזה הוא 4 בתים, לכן אנחנו רוצים לרוץ על הבתים בבלוק הזיכרון עד לרביעי מהסוף, כי בדיקה של שלוש הכתובות אחרי כ-DWORD תוביל לחריגה מגבולות הבלוק.

נניח ובלוק הזיכרון הוא בגודל 5 בתים ונראה כך (כל מלבן סגול הוא בית):

המצביע שלנו יעבור בית בית ויבדוק מהמיקום שלו ערך של 4 בתים:

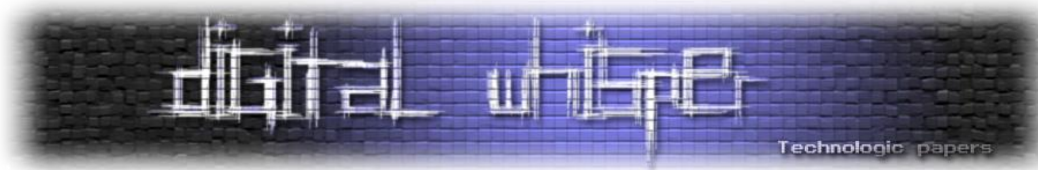
התקדמות של המצביע אל בתים מעבר לבית ה-4 מהסוף לא רצויה:



עכשיו כשעניין הכתובות סגור, אפשר להגיע אל הלולאה. הלולאה תראה כך:

```
while (minAddress <= maxAddress)
{
	if (*(DWORD*)minAddress == LIST_HEAD_MAGIC) Roll(minAddress);
	minAddress = (LPBYTE)minAddress + 1;
}
```

בטוח שאתה זוכר נכון?



בכל פעם אנחנו בודקים את ערך ה-DWORD שנמצא בכתובת minAddress. כדי לקבל ערך DWORD יש להתייחס אל minAddress כמצביע לDWORD באמצעות (*DWORD). הכוכבית הנוספת מביאה את הערך שנמצא בכתובת, אחרת היינו משווים את הכתובת ל-LIST_HEAD_MAGIC וזה לא רצוי. אם אכן מצאנו את הערך שאנחנו מחפשים אנחנו קוראים לפונקציה Roll עם הכתובת הרלוונטית. לאחר מכן מגדילים את minAddress ב-1 כדי להמשיך את פעולת הלולאה.

הפונקציה Roll מקבל את הכתובת של הערך השווה ל-LIST_HEAD_MAGIC בזיכרון. הפונקציה תתייחס אליה כאל הכתובת של dwMagic ב-struct LINKED_LIST_HEAD. נגדיר שני מצביעים:

```
LINKED_LIST_HEAD *ListHead;
LINKED_LIST_MEMBER *Member;
```

ListHead ישמש אותנו להתחלת התהליך והצבעה על רשימת הסיסמאות. Member יצביע בכל פעם על struct מסוג LINKED_LIST_MEMBER ובעזרתו נרוץ על הרשימה. כאמור, משתני ה-struct פרוסים ברצף בזיכרון לפי סדר ההצהרה עליהם, כשכתובת הבסיס של ה-struct היא הכתובת של המשתנה הראשון. המשתנה הראשון ב-struct הוא dwMagic והפונקציה מקבלת את הכתובת שלו כ-LPVOID Address. כלומר, הכתובת שעליה מצביע Address היא הכתובת שעליה נצביע עם ListHead בצורה הבאה:

```
ListHead = (LINKED_LIST_HEAD*)Address;
```

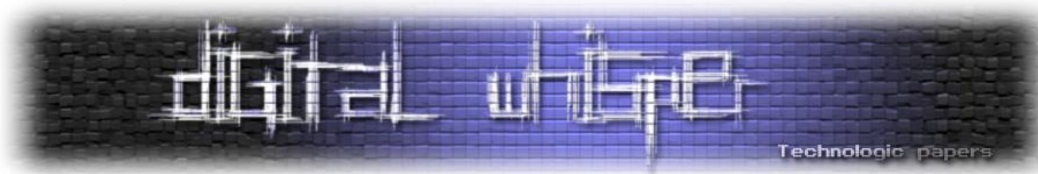
עכשיו כשיש לנו את המצביע לאיבר הראשון ברשימה, נצביע אליו עם Member:

```
Member = ListHead->pFirstMember;
```

נרוץ על הרשימה עם הלולאה הבאה:

```
while (Member->dwPassId != ROCKET_SYSTEM_PASSWORD) Member = Member->Next;
```

כמו שאמרתי, אנחנו מחפשים את הסיסמא למערכת הטילים, לכן, נרוץ עם Member ברשימה עד שנגיע ל-LINKED_LIST_MEMBER עם dwPassId מתאים. כשנצא מהלולאה Member יצביע לאיבר המתאים ברשימה.



כל מה שנשאר הוא לקחת את הסיסמא. נגדיר משתנה ונקצה לו זיכרון בגודל של dwPassLen + 1. dwPassLen בשביל תוכן הסיסמא ו-1 בשביל לסיים את המחרוזת עם null character:

```
char *password;
password = (char*)malloc(Member->dwPassLen + 1);
```

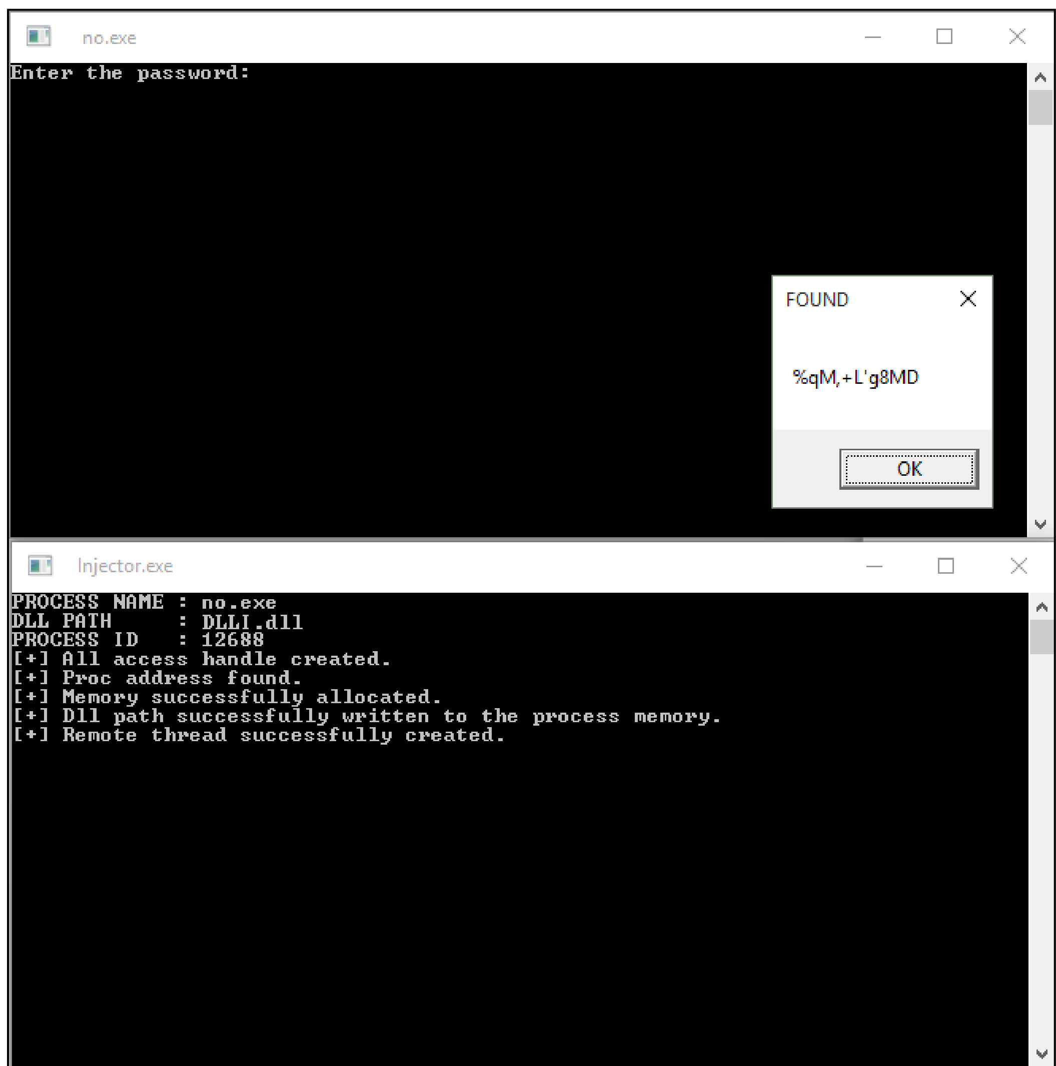
נעתיק את המידע שעליו מצביע lpPassword אל ה-buffer שלנו ונתחום את המחרוזת:

```
memcpy(password, Member->lpPassword, Member->dwPassLen);
password[Member->dwPassLen] = 0;
```

הפרמטר השלישי ב-memcpy הוא כמות הבתים שיש להעתיק. במקרה שלנו זה אורך הסיסמא שמצוין ב-dwPassLen. נקפיץ את הסיסמא כהודעה ובזה מסתיים הקוד:

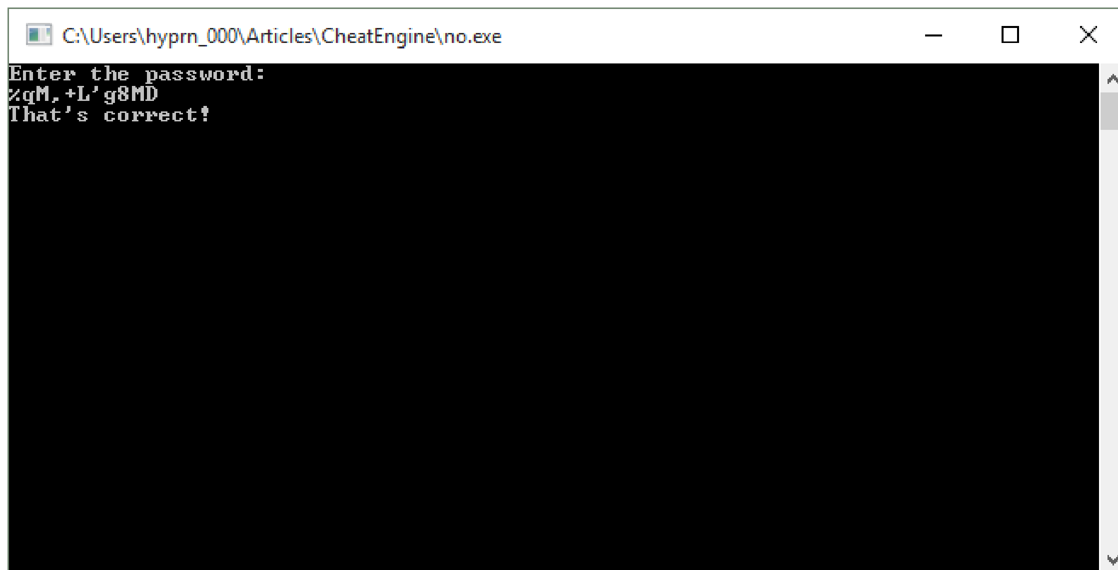
```
MessageBoxA(NULL, password, "FOUND", MB_OK);
```

נריץ ונבדוק את התוצאה. ראשית נריץ את ה-crackme, ולאחר מכן נזריק לו את ה-DLL:



בטוח שאתה זוכר נכון?

נכניס את התוצאה:



זו הייתה הדרך הראשונה. עכשיו אציג את הדרך ללא הזרקת DLL, ואסביר רק את ההבדלים בין הגרסאות.

פיתרון 2: בלי DLL injection

בשיטה זו אנחנו לא נעבוד ישירות מול הזיכרון של התהליך, מכיוון שלכל תהליך מרחב כתובות פרטי משלו אנחנו לא נוכל פשוט לפנות לכתובות ולמשוך מהן ערכים. נאלץ להיעזר בפונקציות WINAPI כדי לגשת לזיכרון התהליך. אנחנו נסרוק את כולו ובזיכרון שנסרוק נחפש את המידע הדרוש. בשביל לנהל את הזיכרון שנסרוק, נשתמש ברשימה מקושרת של struct שיעזור לנו. ה-struct ייצג בלוק זיכרון והמבנה שלו יושפע ישירות מהפרמטרים שמקבלת הפונקציה ReadProcessMemory והם:

```
_In_  HANDLE  hProcess,
_In_  LPCVOID lpBaseAddress,
_Out_ LPVOID  lpBuffer,
_In_  SIZE_T  nSize,
_Out_ SIZE_T  *lpNumberOfBytesRead
```

ה-struct ייראה כך:

```
typedef struct MemoryBlock
{
	HANDLE hProc;
	LPVOID BaseAddress;
	SIZE_T Size;
	char *Buffer;
	struct MemoryBlock *Next;
}MemoryBlock;
```

hProc הוא ה-HANDLE לתהליך ממנו אנחנו קוראים את הזיכרון. BaseAddress הוא המצביע לכתובת הבסיס של הבלוק. Size הוא גודל הבלוק. Buffer הוא ה-buffer בו נאחסן את הזיכרון שנקרא. Next הוא מצביע ל-MemoryBlock הבא ברשימת הבלוקים. פונקציית ה-main שלנו תתחיל בקליטה של שם ה-process מהמשתמש ומציאת ה-PID בדיוק כמו ב-injector שבו השתמשנו קודם. נגדיר משתנה:

```
MemoryBlock *firstBlock;
```

המצביע הזה יצביע לאיבר הראשון ברשימה המקושרת שלנו. נקרא לפונקציה SearchValue שמקבלת pid של תהליך, סורקת אותו ומחזירה מצביע מסוג *MemoryBlock לאיבר הראשון ברשימה מקושרת:

```
firstBlock = ScanProcess(pid);
```

הפונקציה ScanProcess מתחילה עם הגדרת משתנים:

```
SYSTEM_INFO si;
MEMORY_BASIC_INFORMATION mbi;
LPVOID minAddress, maxAddress;
HANDLE hProc;
MemoryBlock *firstBlock = NULL, *Block = NULL;
```

ב-si, mbi, minAddress ו-maxAddress שימוש זהה לשימוש בפיתרון עם ה-DLL. hProc יהיה ה-handle לתהליך ה-crackme. firstBlock יצביע לאיבר הראשון ברשימת איברי ה-MemoryBlock ועם Block נרוץ על הרשימה. נמצא את הערכים ל-minAddress ו-maxAddress. עכשיו נפתח handle לתהליך עם הפונקציה OpenProcess:

```
hProc = OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_READ, NULL, pid);
```

הפרמטר הראשון שמועבר לפונקציה הוא ההרשאות שאנחנו רוצים לקבל על התהליך. במהלך הפיתרון נשתמש בפונקציות שדורשות handle לתהליך. הפונקציות הן VirtualQueryEx ו-ReadProcessMemory. VirtualQueryEx דורשת handle שנפתח עם הרשאות PROCESS_QUERY_INFORMATION ו-ReadProcessMemory דורשת handle שנפתח עם הרשאות PROCESS_VM_READ ולכן אלה הערכים שאנחנו מעבירים ל-OpenProcess. עכשיו נכנס ללולאה שתיראה כך:

```
while (minAddress < maxAddress)
{
	VirtualQueryEx(hProc, minAddress, &mbi, sizeof(mbi));
	if (mbi.State == MEM_COMMIT && mbi.Protect == PAGE_READWRITE)
	{
		if (!firstBlock) {
			firstBlock = CreateMemoryBlock(&mbi, hProc);
			Block = firstBlock;}
		else {
			Block->Next = CreateMemoryBlock(&mbi, hProc);
			Block = Block->Next;}
	}
	minAddress = (LPBYTE)mbi.BaseAddress + mbi.RegionSize;
}
```

הריצה באמצעות minAddress ו-maxAddress זהה לזו שבפיתרון עם ה-DLL. הפעם אנחנו משתמשים ב-VirtualqueryEx במקום VirtualQuery. הן עובדות בצורה זהה, אך VirtualqueryEx מאפשרת עבודה על זיכרון של תהליך אחר.

אנחנו בודקים את ההרשאות על בלוק הזיכרון, במידה והזיכרון בשימוש ומאפשר קריאה וכתיבה ניצור בלוק ונוסיף אותו לרשימה המקושרת שלנו. בתחילת הפונקציה אתחלנו את firstBlock כ-NULL. לכן, אם זה הבלוק המתאים הראשון firstBlock עדיין NULL והרשימה ריקה. ניצור MemoryBlock באמצעות הפונקציה CreateMemoryBlock שמיד אציג ונצביע אליו עם firstBlock. נצביע ל-firstBlock עם Block כדי שנוכל לרוץ על הרשימה מרגע זה ונשמור את ההצבעה על האיבר הראשון באמצעות firstBlock.

במקרה בו firstBlock אינו NULL, Block כבר מצביע על חלק מהרשימה. בין אם על האיבר הראשון או איבר מתקדם יותר. Block->Next יהיה שווה ל-NULL ו-Block יצביע על struct עם מידע חיוני. כדי להמשיך את הריצה על הרשימה ולשמור על המידע החיוני ניצור MemoryBlock חדש ונצביע אליו עם Block->Next. אחרי שיצרנו את ה-MemoryBlock נצביע אל Block->Next עם Block וכך נתקדם ברשימה.

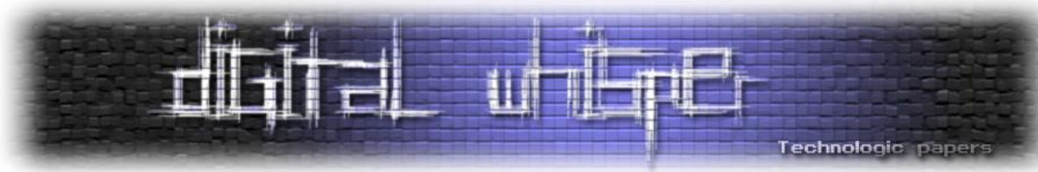
הפונקציה CreateMemoryBlock מקבלת מצביע ל-struct MEMORY_BASIC_INFORMATION ו-handle לתהליך של ה-crackme. שני הפרמטרים האלה מכילים את כל המידע הדרוש ל-struct MemoryBlock.

הפונקציה נראית כך:

```
MemoryBlock* CreateMemoryBlock(MEMORY_BASIC_INFORMATION *mbi, HANDLE hProc)
{
	MemoryBlock *Block = (MemoryBlock*)malloc(sizeof(MemoryBlock));

	Block->hProc = hProc;
	Block->BaseAddress = mbi->BaseAddress;
	Block->Size = mbi->RegionSize;
	Block->Buffer = (char*)malloc(mbi->RegionSize);
	Block->Next = NULL;

	return Block;
}
```

בפונקציה אנחנו רק מקצים זיכרון לbuffer ולא מבצעים קריאה של הזיכרון, אין סיבה מיוחדת לכך. נחזור ללולאה. הלולאה מסתיימת בעדכון של minAddress בדיוק כמו בפתרון עם הDLL. לאחר סיום הריצה של הלולאה אמורה להיות לנו רשימה מקושרת של struct מסוג MemoryBlock שמכילה את המידע שצריך כדי לקרוא את הזיכרון מהתהליך. נחזיר את המצביע לראש הרשימה ונחזור לmain.

עכשיו כשאנחנו שוב בmain נקרא לפונקציה ReadMemory שמקבלת מצביע ל-struct MemoryBlock, עוברת על כל האיברים ברשימה וקוראת אל המשתנה Buffer את הזיכרון שאותו הם מייצגים.



הפונקציה ReadMemory מתחילה מהגדרת מצביע מקומי לאיבר הראשון ברשימה כדי לא לשנות את ערך המצביע שמועבר אליה:

```
MemoryBlck *Block = firstBlock;
```

איברים ברשימה נוצרים כך שהאיבר הבא אחרים הוא NULL, לכן האיבר האחרון ברשימה הוא NULL. כדי לעבור על כל האיברים אנחנו נשתמש בלולאה הבאה:

```
while (Block)
{
   ReadProcessMemory(Block->hProc, Block->BaseAddress, Block->Buffer, Block->Size, NULL);
   Block = Block->Next;
}
```

כל עוד Block אינו NULL נקרא זיכרון באמצעות ReadProcessMemory אל המשתנה Buffer ב-struct MemoryBlock ונתקדם ברשימה. אין צורך בהחזרה של מצביע. הקריאה התבצעה על הרשימה שלנו מפני שהעברנו לפונקציה מצביע לרשימה ולא העתק שלה ולכן כל השינויים נעשו על הרשימה המקורית.

עכשיו כשקראנו את כל הזיכרון הגיע הזמן לחפש את הערך LIST_HEAD_MAGIC. נקרא לפונקציה SearchValue שבמקרה הזה מקבלת מצביע לאיבר הראשון ברשימת MemoryBlock וערך שאותו היא תחפש (מעט שונה מהגרסא עם ה-DLL.):

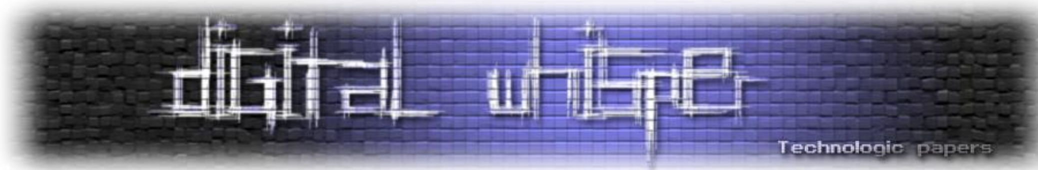
```
pValue = SearchValue(firstBlock, LIST_HEAD_MAGIC);
```

SearchValue דומה מאוד בשני הפתרונות. הפעם במקום לבדוק בלוק אחד שהיא מקבלת כ-MEMORY_BASIC_INFORMATION היא רצה על רשימה של MemoryBlock ובודקת את ה-Buffer שלהם. נגדיר מצביע מקומי לאיבר הראשון ברשימה מאותה הסיבה שעשינו זאת ב-ReadMemory ומשתנה int שישמש אותנו ללולאת for. נרוץ על הרשימה באופן זהה לריצה ב-ReadMemory אך הפעם הפעולה על כל MemoryBlock תהיה שונה כמובן:

```
while (Block)
{
   for (i = 0; i <= Block->Size - sizeof(DWORD); i++)
   {
      if (*(DWORD*)(Block->Buffer + i) == value)
      return (LPVOID)((LPBYTE)Block->BaseAddress + i);
   }
   Block = Block->Next;
}
```

כל עוד Block שונה מ-NULL לא הגענו לסוף הרשימה והריצה ממשיכה. עבור כל בלוק נשתמש בלולאת for כדי לסרוק את הBuffer שלו בחיפוש אחר LIST_HEAD_MAGIC (שהועבר לפונקציה כ-value).

ההגדרה i <= Block->Size - sizeof(DWORD) נועדה למנוע חריגת זיכרון כמו במקרה הקודם. גם כאן נבדוק עבור כל כתובת את ערך ה-DWORD שעליו היא מצביעה. במקרה ומצאנו כתובת המצביעה על הערך LIST_HEAD_MAGIC נחזיר את הכתובת.



עכשיו כשיש לנו את הכתובת של LIST_HEAD_MAGIC אפשר לקרוא ל-Roll. במקרה שלנו Roll מקבלת את הכתובת של LIST_HEAD_MAGIC ואת ה-handle לתהליך ה-crackme. היא זקוקה לו כדי לבצע קריאות זיכרון נוספות מהתהליך, למרות שהמידע כבר אצלנו במשתני הBuffer ברשימה הקוד פשוט יותר כך. הפונקציה מתחילה עם הגדרת המשתנים הבאים:

```
PASSWORD_LINKED_LIST_POINTER *ListPointer =
(PASSWORD_LINKED_LIST_POINTER*)malloc(sizeof(PASSWORD_LINKED_LIST_POINTER));

PASSWORD_LINKED_LIST_MEMBER *Member =
(PASSWORD_LINKED_LIST_MEMBER*)malloc(sizeof(PASSWORD_LINKED_LIST_MEMBER));

char *pass;
```

ListPointer הוא המצביע לאיבר הראשון ברשימה שמכיל את dwMagic. Member יאפשר לנו לרוץ על הרשימה ולחפש את הסיסמא. עד עכשיו הרשימה שעליה דיברנו היא רשימת MemoryBlock, בפונקציה Roll הרשימה היא רשימת הסיסמאות ששמורות ב-struct PASSWORD_LINKED_LIST_MEMBER. ב-pass נשמור את הסיסמא שנמצא. נקרא זיכרון מהתהליך בכתובת של LIST_HEAD_MAGIC:

```
ReadProcessMemory(hProc, pValue, ListPointer, sizeof(PASSWORD_LINKED_LIST_POINTER),
NULL);
```

אנחנו קוראים זיכרון בגודל ה-struct PASSWORD_LINKED_LIST_POINTER מהכתובת של LIST_HEAD_MAGIC. LIST_HEAD_MAGIC הוא הערך שיושב בdwMagic הלא הוא המשתנה הראשון ב-struct PASSWORD_LINKED_LIST_POINTER. כפי שהסברתי, כתובת הבסיס של struct היא הכתובת של המשתנה הראשון שלו. לכן, בהנחה והכתובת שיש לנו היא הכתובת הנכונה, קריאת זיכרון ממנה בגודל של ה-struct הרצוי היא בעצם קריאה של ה-struct. עכשיו כשיש לנו את המצביע לרשימה, נגדיר את האיבר הראשון ברשימה.

בשביל זה נקרא זיכרון בצורה דומה, הפעם לMember. הכתובת שממנה נקרא היא הכתובת שנמצאת ב-ListPointer->FirstMember. גם הפעם נקרא זיכרון בגודל המתאים ל-struct:

```
ReadProcessMemory(hProc, ListPointer->FirstMember, Member,
sizeof(PASSWORD_LINKED_LIST_MEMBER), NULL);
```

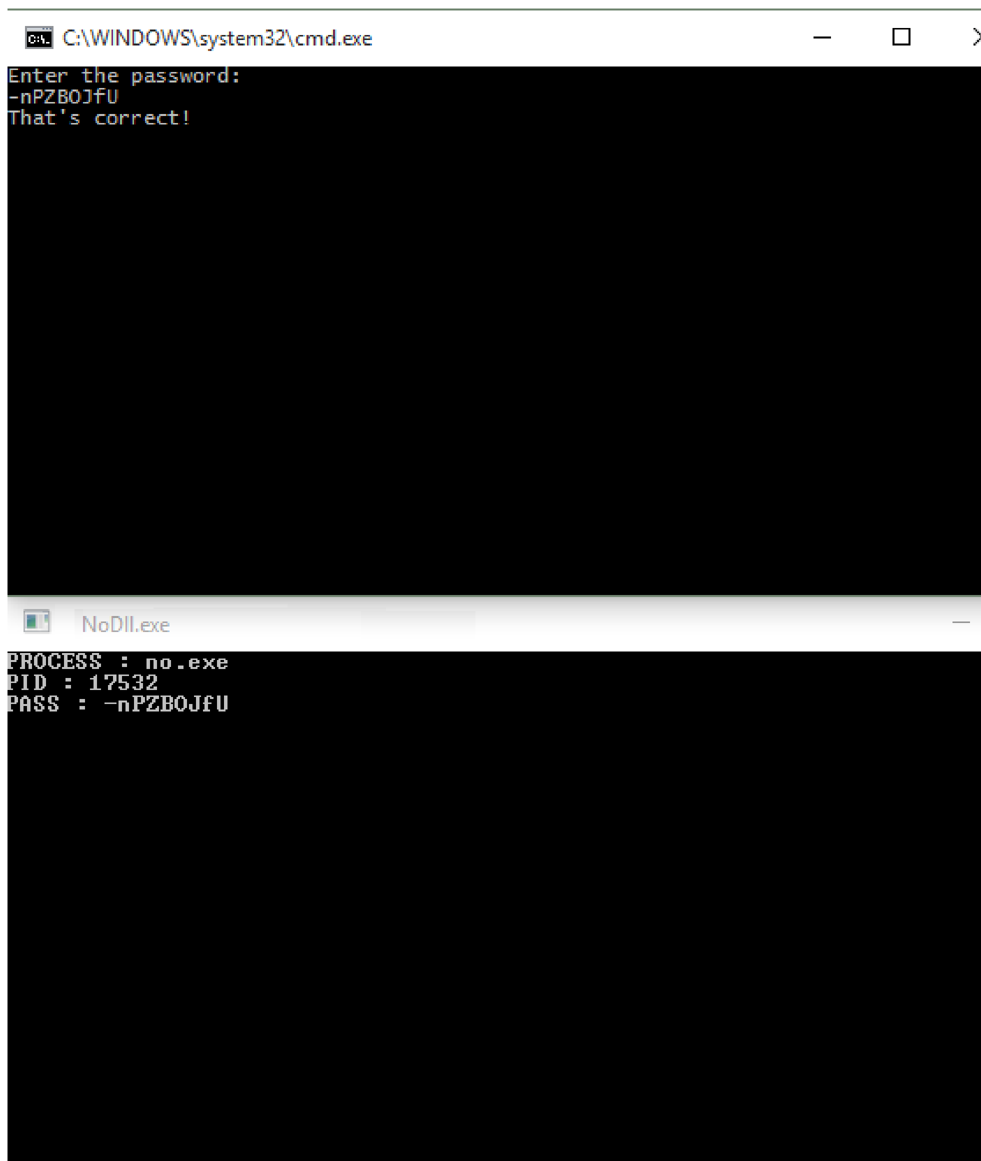
עכשיו כשיש לנו את האיבר הראשון ברשימה אפשר להתחיל לרוץ עליו ולחפש את הסיסמא שהdwPassId שלה הוא ROCKET_SYSTEM_PASSWORD, הרי אנחנו מחפשים את הסיסמא למערכת הטילים.

כל עוד האיבר ברשימה שאליו אנחנו מצביעים עם Member לא מאחסן את הסיסמא המתאימה נתקדם על ידי קריאת זיכרון מהתהליך בכתובת של Member->NextMember אל Member.

ברגע ש-Member->dwPassId שווה ל-ROCKET_SYSTEM_PASSWORD הגענו לסיסמא שאנחנו מחפשים. נקצה ל-pass זיכרון בגודל Member->dwPassLen ונקרא אליו את הסיסמא. בנוסף, נתחום את המחרוזת עם null character. לאחר מכן נדפיס את הסיסמא:

```
while (Member->dwPassId != ROCKET_SYSTEM_PASSWORD) ReadProcessMemory(hProc, Member->NextMember, Member, sizeof(PASSWORD_LINKED_LIST_MEMBER), NULL);

pass = (char*)malloc(Member->dwPassLen + 1);

ReadProcessMemory(hProc, Member->lpPassword, pass, Member->dwPassLen, NULL);
pass[Member->dwPassLen] = 0;

printf("PASS : %s\n", pass);
```

נריץ את ה-crackme ולאחר מכן את הקוד שכתבנו:



זהו. פתרנו את ה-crackme בשתי דרכים, עם הזרקת DLL וללא הזרקת DLL. ההקדמה כנראה גרמה לקוד להזכיר Cheat Engine, אבל הפונקציות שכתבנו מאפשרות בסיס לקוד להיות הרבה יותר קרוב לתוצאה הסופית ממה שנדמה.

Cheat Engine

נניח בצד את פתרון ה-crackme עם ה-DLL ונבחן את הפתרון השני. באמצעות הפונקציות שכתבנו בו וה-struct MemoryBlock יש לנו את היכולת לסרוק זיכרון של תהליך, ולמצוא בו מיקומים של ערכי DWORD. בתחילת המאמר דיברתי על כך ששימוש נפוץ ב-Cheat Engine הוא שינוי של ערכים מספריים במשחקים. עם הקוד שיש לנו אנחנו יכולים למצוא את הערכים האלה. כל שנשאר הוא לשנות אותם. נפתח פרויקט חדש, נעתיק אליו את ה-struct MemoryBlock ואת הפונקציות CreateMemoryBlock, ScanProcess, ReadMemory ו-GetPid (כתובה ב-injector).

נכתוב את main. התחלה, כמו תמיד, עם הגדרת משתנים:

```
DWORD pid, Value, newValue, cheatValue;
char *pName, *input;
MemoryBlock *firstBlock;
Location *firstLocation;
```

pid ישמש אותנו למציאת התהליך אותו נסרוק. Value הוא הערך שנחפש ונרצה לשנות. newValue יאפשר לנו לצמצם את רשימת הערכים שנמצא על ידי הכנסה של ערך נוסף והשוואה שלו לערכים שנמצאים בכתובות שמצאנו בחיפוש ראשוני. cheatValue יכיל את הערך שנכניס כדי להחליף את הערך המקורי.

pName יכיל את שם התהליך שנסרוק. Input ישמש אותנו לקליטת קלט מהמשתמש. firstBlock יצביע על האיבר הראשון ברשימה של MemoryBlock. firstLocation יצביע על האיבר הראשון ברשימה של Location, struct שנגדיר עוד מעט.

נתחיל ממציאת ה-pid של התהליך לפי שם שהמשתמש מכניס:

```
pName = (char*)malloc(MAX_PATH + 1);
fgets(pName, MAX_PATH, stdin);
pName[strlen(pName) - 1] = 0;
pid = GetPid(pName);
```

נסרוק את התהליך ונקרא את הזיכרון שלו:

```
firstBlock = ScanProcess(pid);
ReadMemory(firstBlock);
```

הפעם ביצירת ה-handle לתהליך בפונקציה ScanProcess נקרא ל-OpenProcess בצורה שונה. בכלי הזה ברצוננו לכתוב לזיכרון התהליך כדי לשנות בו ערכים מסוימים. בשביל כתיבה לזיכרון התהליך נשתמש בפונקציה WriteProcessMemory שמקבלת כפרמטר handle לתהליך שנפתח עם הרשאות PROCESS_VM_WRITE וPROCESS_VM_OPERATION. הקריאה ל-OpenProcess תיראה כך:

```
hProc = OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_READ | PROCESS_VM_WRITE
| PROCESS_VM_OPERATION, NULL, pid);
```

אנחנו משאירים את PROCESS_QUERY_INFORMATION וPROCESS_VM_READ בשביל סריקת זיכרון התהליך. אפשר להחליף את קבוצת ההרשאות הזו ב-PROCESS_ALL_ACCESS והפעולות שנעשה עדיין יעבדו. נקלוט מהמשתמש את ערך ה-DWORD שיש למצוא:

```
printf("DWORD VALUE : ");
input = (char*)malloc(MAX_PATH + 1);
fgets(input, MAX_PATH, stdin);
sscanf(input, "%d", &Value);
```

אין סיבה מיוחדת לשימוש ב-MAX_PATH, אני בדרך כלל משתמש בו לקלט מחרוזות. הערך שלו הוא 260, האורך המקסימאלי לנתיב קובץ ב-Windows. עכשיו נגדיר את ה-struct Location, מבנה פשוט של רשימה מקושרת של מצביעים. כשנחפש ערך בזיכרון התהליך, נשמור את ההופעות שלו ברשימה כזאת:

```
typedef struct Location
{
	LPVOID Address;
	struct Location *Next;
}Location;
```

ופונקציה ליצירת Location:

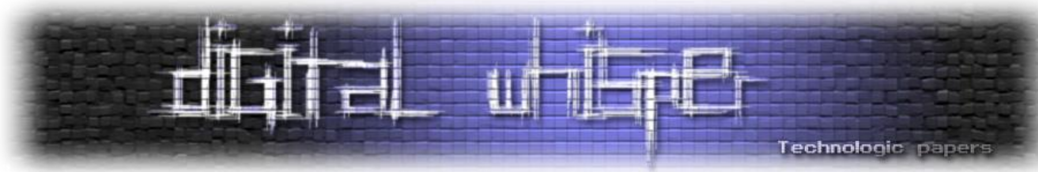
```
Location* CreateLocation(LPVOID Address)
{
	Location *l = (Location*)malloc(sizeof(Location));
	l->Address = Address;
	l->Next = NULL;
	return l;
}
```

נקרא לפונקציה FindLocations, המקבילה לSearchValue במקרה שלנו, שמקבלת ערך DWORD ומצביע לאיבר ראשון ברשימת MemoryBlock. הפונקציה מחזירה מצביע לאיבר הראשון ברשימת Location:

```
firstLocation = FindLocations(firstBlock, Value);
```

נגדיר את המשתנים שישמשו אותנו בפונקציה:

```
Location *firstLocation = NULL, *tempLocation = NULL;
MemoryBlock *Block = firstBlock;
int i = 0;
```



firstLocation יצביע על האיבר הראשון ברשימת Location שנחזיר, וtempLocationו ישמש אותנו בריצה עליה. Block הוא מצביע מקומי לאיבר הראשון ברשימת MemoryBlock שמאפשר לנו לרוץ על הרשימה בלי לפגוע במבנה שלה. i ישמש אותנו בסריקה של Buffer בכל MemoryBlock. הפונקציה כולה תתנהל בלולאת while, בה נרוץ על רשימת ה-MemoryBlock ונחפש אחר הערך Value:

```
while (Block)
{
   for (i = 0; i <= Block->Size - sizeof(DWORD); i++)
   {
      if (*(DWORD*)(Block->Buffer + i) == Value)
      {
         if (!firstLocation)
         {
            firstLocation = CreateLocation((LPBYTE)Block->BaseAddress + i);
           templocation = firstLocation;
         }
         else {
            templocation->Next = CreateLocation((LPBYTE)Block->BaseAddress + i);
            templocation = templocation->Next;
         }
      }
   }
   Block = Block->Next;
}
```

כל עוד Block אינו NULL לא הגענו לסוף רשימת ה-MemoryBlock וזהו MemoryBlock שיש לסרוק את ה-Buffer שלו. נרוץ על ה-Buffer בלולאת for, את המבנה שלה הסברתי בשני הפתרונות ל-crackme. במידה ומצאנו את הערך Value נבדוק האם זו הפעם הראשונה שמצאנו אותו. במידה וזו הפעם הראשונה, firstLocation הוא NULL. ניצור Location חדש ונצביע עליו עם firstLocation ועם templocation, את ההצבעה של firstLocation לא נשנה יותר. אם firstLocation אינו NULL זו אינה הפעם הראשונה שמצאנו את הערך. ניצור Location חדש ונצביע עליו עם templocation->Next. לאחר מכן נצביע על templocation->Next עם templocation כדי להתקדם ברשימה. לאחר מכן נתקדם ברשימת הMemoryBlock באמצעות הצבעה על Block->Next עם Block.

לאחר שסיימנו לרוץ על רשימת ה-MemoryBlock נחזיר את המצביע לאיבר הראשון ברשימת ה-Location. נדפיס את תוכן רשימת ה-Location עם הפונקציה PrintLocationsValues שמקבלת מצביע לאיבר הראשון ברשימת Location ו-handle לתהליך הנסרק. הפונקציה תדפיס את הכתובת שבה הערך נמצא, תקרא את הכתובת שוב ותדפיס את הערך שנמצא בה (במקרים מסוימים הערך בכתובת מספיק להשתנות):

```
PrintLocationsValues(firstLocation, firstBlock->hProc);
```

נגדיר מצביע מקומי לאיבר הראשון ברשימת ה-Location ומשתנה DWORD שאליו נקרא את הערך שנמצא בכתובת שמצאנו:

```
Location *l = firstLocation;
DWORD value = 0;
```

כל עוד l אינו NULL נקרא את הערך בכתובת, נדפיס אותו ואת הכתובת ונתקדם:

```
while (l)
{
	ReadProcessMemory(hProc, l->Address, &value, sizeof(DWORD), NULL);
	printf("ADDR : 0x%p\tValue : %d\n", l->Address, value);
	l = l->Next;
}
```

בחזרה ל-main. נקלוט מהמשתמש ערך DWORD נוסף. הערך הזה נועד לסינון התוצאות שנמצאות ברשימת ה-Location. למשל, למשתמש יש 1,000,000 מטבעות במשחק. המשתמש מכניס בפעם הראשונה את הערך 1,000,00. המשתמש משאיר את הכלי פתוח ומשנה את כמות המטבעות שלו על ידי רכישה כלשהי או רווח מסוים וכעת יש לו 900,000 מטבעות. המשתמש יכניס את הערך 900,000 והכלי יתמקד בכתובות שבהתחלה הכילו את הערך 1,000,000 וכעת מכילות את הערך 900,000. סיכוי גדול שאלה הן הכתובות שמעניינות את המשתמש:

```
printf("NEW VALUE : ");
input = (char*)malloc(MAX_PATH + 1);
fgets(input, MAX_PATH, stdin);
sscanf(input, "%d", &newValue);
```

כעת נדפיס את הכתובות בהן מופיע הערך החדש שהכילו בהתחלה את הערך הראשון. נקרא לפונקציה PrintLocationsValuesNew שדומה מאוד ל-PrintLocationsValues אך משווה את הערכים בכתובות שעליהן היא עוברת ומדפיסה אותם רק אם הם שווים לערך החדש. יש לשים לב שרשימת הLocation שעליה עוברות שתי הפונקציות זהה ולא עברה שום שינוי. נקרא לפונקציה:

```
PrintLocationsValuesNew(firstLocation, firstBlock->hProc, newValue);
```

הפונקציה החדשה נראית כך:

```
Location *l = firstLocation;
DWORD value;

while (l)
{
	ReadProcessMemory(hProc, l->Address, &value, sizeof(DWORD), NULL);
	if (value == newValue) printf("ADDR : 0x%p\tValue : %d\n", l->Address, 
value);
	l = l->Next;
}
```

כאמור, התוספת היחידה היא ההשוואה בין value ל-newValue. הגיע רגע האמת, נקלוט מהמשתמש את הערך החדש שיחליף את הערך הקיים ונבצע את הדריסה שלו. נתחיל מקליטת הערך החדש:

```
printf("CHEAT VALUE : ");
input = (char*)malloc(MAX_PATH + 1);
fgets(input, MAX_PATH, stdin);
sscanf(input, "%d", &cheatValue);
```

נקרא לפונקציה שתדרוס את הערכים המקוריים:

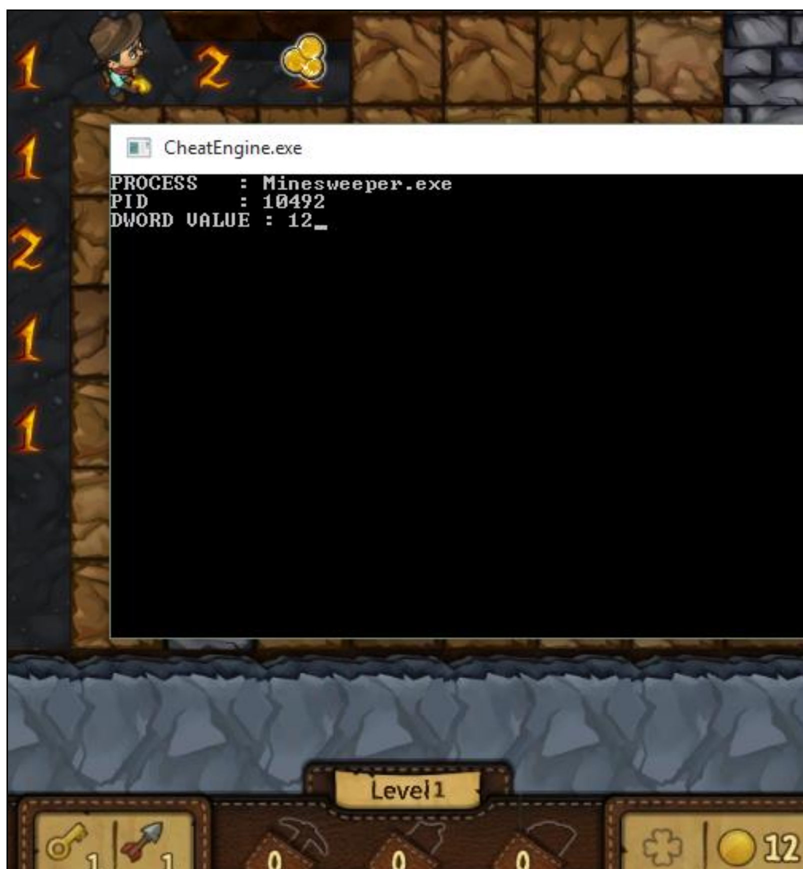
```
WriteNewValue(firstLocation, firstBlock->hProc, newValue, cheatValue);
```

הפונקציה דומה מאוד ל-PrintLocationsValuesNew אך הפעם אם קיים שוויון היא דורסת את הערך המקורי עם cheatValue. הפונקציה נראית כך:

```
Location *l = firstLocation;
DWORD value;
while (l)
{
	ReadProcessMemory(hProc, l->Address, &value, sizeof(DWORD), NULL);
	if (value == newValue) WriteProcessMemory(hProc, l->Address,
(LPVOID)&cheatValue, sizeof(DWORD), NULL);
	l = l->Next;
}
```

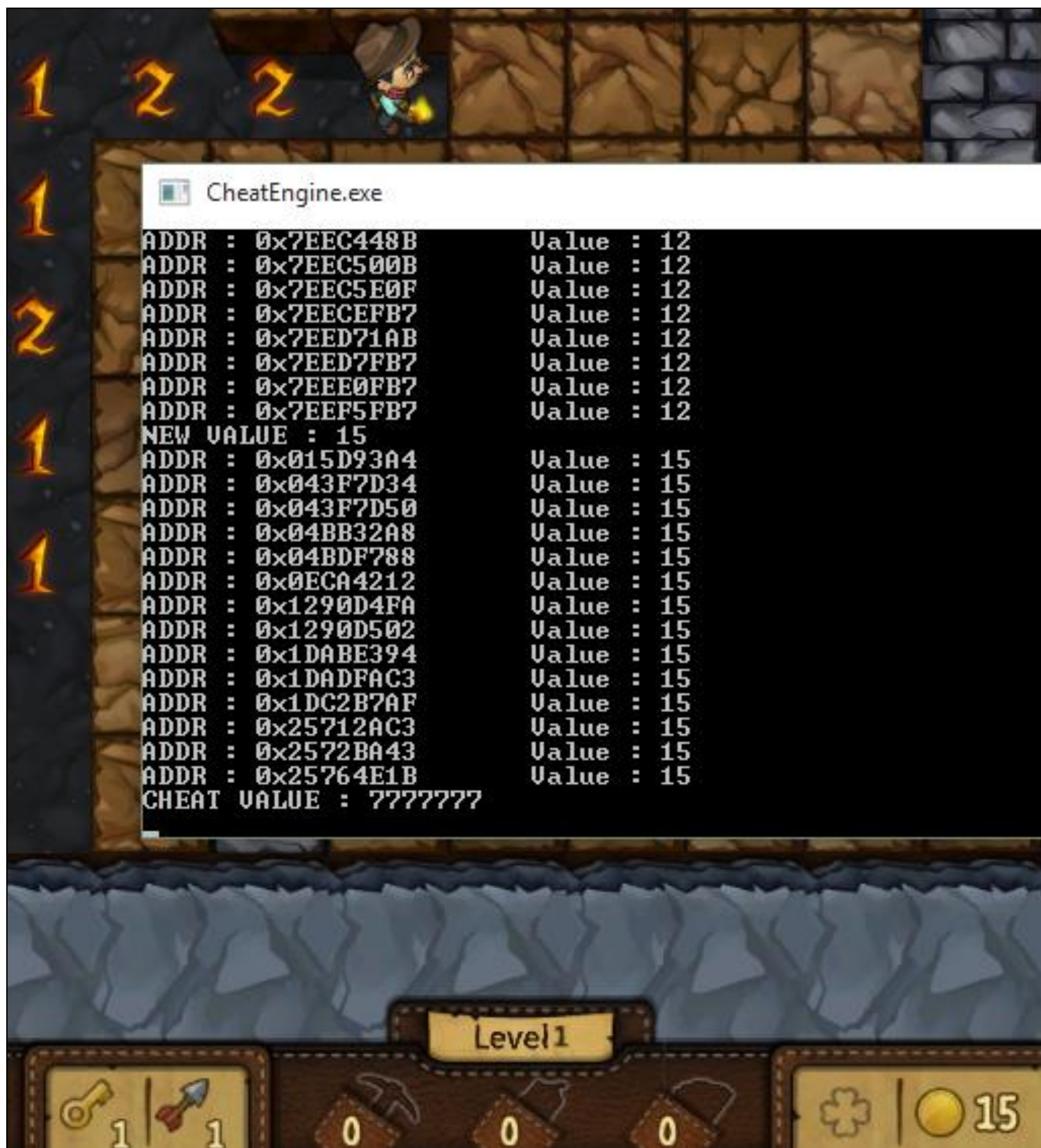
הדריסה של הערך המקורי נעשית באמצעות הפונקציה WriteProcessMemory שמקבל handle לתהליך, כתובת שאליה יש לכתוב, את הערך שיש לכתוב וגודל הערך שברצוננו לכתוב. הפרמטר האחרון מחזיר את כמות הבתים שנכתבו, נתעלם ממנו (במצב תקין הוא צריך להיות זהה לערך שהעברנו לפונקציה המציין את גודל הערך שברצוננו לכתוב).

הכלי מוכן. נבדוק אותו על מוד ה-Adventure של שולה המוקשים. נכנס למשחק ונריץ את הכלי:



יש לנו 12 מטבעות. בכלי נבחר את התהליך Minesweeper.exe, תהליך המשחק. נכניס את הערך הראשוני שברצוננו לחפש, 12. לאחר שהכנסנו את הערך 12 הכלי ידפיס את הכתובות שבו מצא את הערך. יש המון כתובות כאלה. כדי לצמצם את הכתובות נאסוף את המטבעות שנמצאים לידנו:

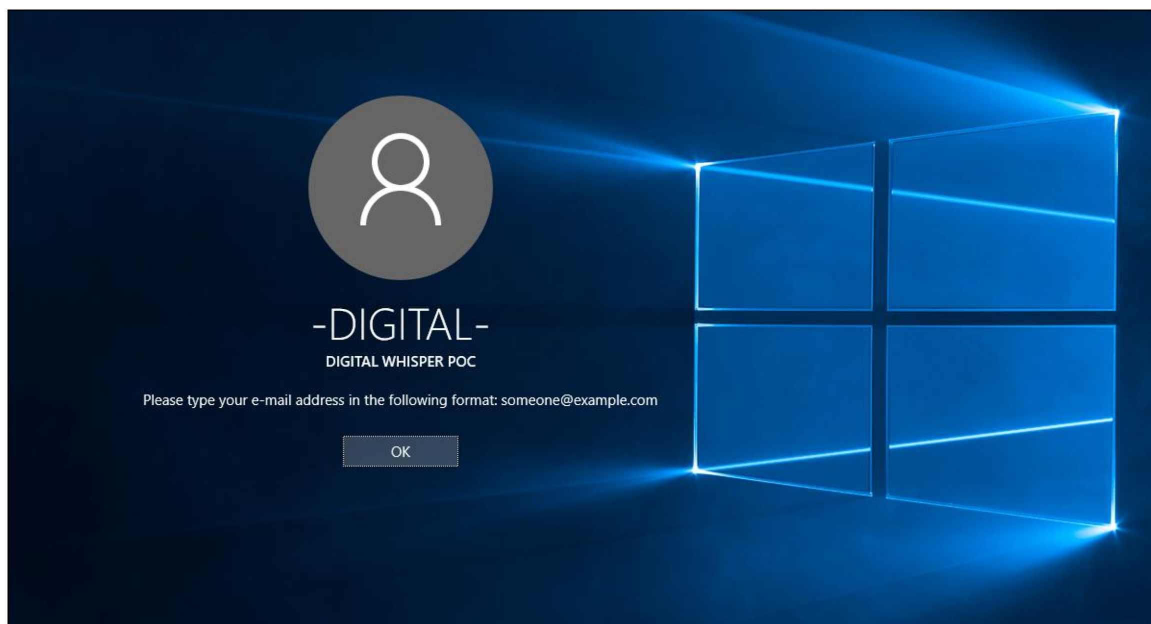
עכשיו כשיש לנו 15 מטבעות נכניס את את הערך 15. הכלי יעבור על הכתובות שבהן מצא את הערך 12 וידפיס רק את אלו שעכשיו מכילות את הערך 15:



גם עכשיו יש מספר גדול יחסית של תוצאות, אך משמעותית קטן יותר מהתוצאות של הסריקה הראשונה שהניבה עשרות תוצאות. נכניס את הערך האחרון. זוהי כמות המטבעות שאנחנו רוצים. כאמור, ההתעסקות הזו היא בעיקר ניסוי וטעיה ויכולה להקריס את התהליך. בזה הסתיימה פעולת הכלי. למרות שעל המסך עדיין מוצג הערך 15 זו אינה באמת כמות המטבעות שיש לנו במשחק. כמות המטבעות כאמור שמורה כערך DWORD בזיכרון התהליך וההצגה של המטבעות על המסך היא כנראה מחרוזת. כדי לראות את השינוי בכמות המטבעות נאסוף עוד קצת מטבעות במשחק. המשחק יחשב את סכום כמות המטבעות שיש לנו (ערך ה-DWORD) וכמות המטבעות שנאסוף:

רימינו את המשחק ☺. נוכל להוסיף גם פצצות, פסילות וכל דבר שניתן לצבור במשחק.

אפשר גם להוסיף אפשרות לשינוי של ערכים מסוגים שונים, מחרוזות למשל:



סיכום

בכתבה אמנם חיפשנו ערכי DWORD אך אפשר בקלות לחפש כל סוג ערך אחר (נסו את הפונקציה memcmp). הכלי שפיתחנו בעל פוטנציאל חזק ואידיאלי לערכים גלויים למשתמש. ניתן לקחת את הליבה של הכלי ולשכלל אותו כך שיתאים לצרכים אישיים. אני באופן אישי אשתמש בו בכמה משחקים, למרות שממוד ה-Adventure של שולה המוקשים הוא הוציא לי את הכיף. ניתן ליצור קשר במייל hyprnir@gmail.com.

תודות

- תודה ל**דור אריאלי** שעזרה לי בכתיבה ונתנה לי ביקורת כנה.
- תודה ל**דימה פשול** שכתב את ה-crackme ועזר לי בכתיבת המאמר.