



گزارش پروژه کارشناسی

Memory Scanner Tool

محسن پاکزاد

استاد پروژه: دکتر تحیری

تعریف پروژه

هدف از انجام این پروژه، نوشتن ابزاری است که با آن بتوان حافظه پروسه (process) های موجود و در حال اجرای کامپیوتر را اسکن کرد و سپس اقدام به خواندن و نوشتن مقادیر دلخواه در این حافظه ها کرد.

بدین صورت که ما فارغ از چهارچوب هایی که توسط برنامه تعیین شده اند می توانیم بر اساس فاکتور مورد نظرممان اقدام به اسکن حافظه یک پروسه کنیم و به صورت مستقیم اقدام به نوشتن یا خواندن در خانه های حافظه آن کنیم.

اگر تا به حال اسم برنامه [Cheat Engine](#) شنیده باشید و با کار کرده باشید، ابزار تولید شده در این پروژه را می توان نسخه ساده شده چیت انجین در نظر گرفت.

برای اسم محصول نوشته شده در این پروژه [Urule](#) در نظر گرفته شده که گرفته شده از جمله "You rule" است که به معنی است که "شما حکم فرمایی می کنید".

برنامه نوشته شده تحت سیستم عامل ویندوز ([Windows](#)) می باشد و برای اجرا و کامپایل کد برنامه نیازمند استفاده از این سیستم عامل است.

نحوه کار با پروژه

برنامه به صورت کلی از سه ایده: اسکن کردن، خواندن و نوشتن در حافظه مربوط به پروسه ها استفاده می‌کند.

بدین صورت که ما توسط قسمت اسکن کننده، به دنبال مقدار دلخواه و در صورت (type) مورد نظرمون در پروسه انتخاب شده می‌گردیم و در حین این عمل، برنامه با توجه به نوع اسکنی که انتخاب کردیم، حافظه مربوط به پروسه انتخاب شده را خوانده و سپس حافظه‌هایی که با شرایط اسکن ما مطابقت داشتند را به ما نشان می‌دهند و در این مرحله می‌توانیم مقادیر دلخواه خود را در این قسمت‌های حافظه بنویسیم.

چون معمولاً تعداد حافظه‌های پیدا شده بعد از یک بار اسکن کردن زیاد هستند، بهتر است قبل از نوشتن مقادیر دلخواه در حافظه‌های انتخاب شده، عمل اسکن چند بار تکرار شود تا هم دقت حافظه‌های پیدا شده بالاتر رود و هم احتمال دچار مشکل شدن (crash کردن) آن پروسه بر اثر عوض شدن مقادیر آن کمتر شود.

در ادامه مورد قبل این نکته را در نظر بگیرید که عوض کردن تعداد قابل توجهی از حافظه‌های پروسه با مقادیر دلخواه، ممکن است در روند آن خلل ایجاد کند و بعضاً پروسه به کلی دچار مشکل شده و متوقف گردد.

جزئیات اسکن

اسکن کردن در واقع چیزی نیست، جز جست و جوی حافظه پروسه برای پیدا کردن خانه هایی از حافظه که شرایطی را که ما می‌خواهیم داشته باشند.

اسکن به خودی خود دارای دو خصوصیت است:

1. نوع خود اسکن
 2. تایپ (صورت) مقداری که در حال جست و جوی آن هستیم
- که در ادامه به توضیحات راجب به هر کدام می‌پردازیم.

نوع اسکن:

نوع اسکن، مشخص کننده و نشان داده شرایط ای هست که ما می‌خواهیم حافظه های پروسه را به اساس آن جست و جو کنیم.

در ادامه، لیست انواع اسکن های پشتیبانی شده آورده شده اند:

1. مقدار دقیق (**Exact value**)
2. مقدار کوچکتر از مقدار داده شده (**Smaller than value**)
3. مقدار بزرگتر از مقدار داده شده (**Bigger than value**)
4. مقدار بین دو بازه (**Value between**)
5. مقدار اولیه نامعلوم (**Unknown initial value**)
6. مقدار زیاد شده (**Increased value**)
7. مقدار زیاد شده به مقدار داده شده (**Increased value by**)
8. مقدار کم شده (**Decreased value**)
9. مقدار کم شده به مقدار داده شده (**Decreased value by**)
10. مقدار تغییر کرده (**Changed value**)
11. مقدار تغییر نکرده (**Unchanged value**)

لازم به ذکر است که اسکن شماره 5 (مقدار اولیه نامعلوم)، همان طور که از نام اش هم مشخص است، فقط برای اولین اسکن قابل استفاده است و برای اسکن های بعدی به خودی خود معنی ای ندارد.

و همین طور برای اسکن های شماره 6 تا 11 نیز، می‌بایست ابتدا مقادیری پیدا شده باشد که سپس بخواهیم به جست و جو در رابطه با مقادیر کم شده، زیاد شده و یا تغییر نکرده و ... نسبت به آن بپردازیم. بر اساس این توضیحات هم این اسکن ها از اسکن دوم به بعد قابل استفاده هستند و برای اولین اسکن به خودی خود معنی ای ندارند.

تایپ (صورت) مقادیر مورد جست و جو:

همان طور که می‌دانیم، حافظه کامپیوتر به خودی خود تشکیل شده از قرارگیری تعدادی بیت (bit) است که به عنوان کوچکترین واحد ذخیره سازی در کامپیوتر های معمول از آن یاد می‌شوند و می‌تواند دو مقدار **صفر** و **یک** را در خود نگه دارد.

در ادامه از آن جایی که ممکن است بخواهیم بازه بیشتری از صفر تا یک را در برنامه نشان دهیم، با قرار دادن تعدادی بیت در کنار دیگر می‌توانیم به این هدف برسیم، مثلاً بسته به نیاز، می‌توانیم اعداد با بازه های مختلف را از قرارگیری 8، 16، 32 و ... بیت در کنار هم دیگر داشته باشیم.

حال که دیدیم حافظه در واقع حاصل قرارگیری تعدادی بیت کنار هم دیگر است و مقادیری هم که داخل حافظه هستند معمولاً از قرارگیری چند بیت در کنار هم دیگر برداشت می‌شوند، در اینجا ما نیاز به یک **پیمانه** داریم تا مشخص کنیم با قرارگیری هر چند بیت در کنار هم می‌خواهیم به مقدار نشان دهنده آن‌ها نگاه کنیم.

این پیمانه‌ها در کنار اینکه ممکن است سایز متفاوتی داشته باشند، حتی می‌توانند به صورتی که به حاصل قرار گیری بیت‌ها در کنار هم نگاه می‌کنند متفاوت باشد، مثلاً برای نشان دادن اعداد صحیح علامت دار از روش مکمل دو (two's complement) یا برای نشان دادن اعداد اعشاری از استاندارد های ذخیره سازی خاص خود استفاده می‌شود.

در معماری های کامپیوتر های معمول، برای راحتی استفاده و اینکه چون معمولاً بیت های قرار گرفته در کنار هم تعداد زیادی دارند به جای واحد بیت از بایت (byte) استفاده می‌شود که هر **هشت بیت** را یک **بایت** می‌نامیم.

در ادامه تایپ های پشتیبانی شده با سایز ها و نگاه های مختلف آورده شده اند:

- 8 بیت یا 1 بایت، عدد صحیح با نگاه علامت دار (**I8 (Signed Byte)**)
- 8 بیت یا 1 بایت، عدد صحیح با نگاه بدون علامت (**U8 (Unsigned Byte)**)
- 16 بیت یا 2 بایت، عدد صحیح با نگاه علامت دار (**I16 (Signed 2 Bytes)**)
- 16 بیت یا 2 بایت، عدد صحیح با نگاه بدون علامت (**U16 (Unsigned 2 Bytes)**)
- 32 بیت یا 4 بایت، عدد صحیح با نگاه علامت دار (**I32 (Signed 4 Bytes)**)
- 32 بیت یا 4 بایت، عدد صحیح با نگاه بدون علامت (**U32 (Unsigned 4 Bytes)**)
- 64 بیت یا 8 بایت، عدد صحیح با نگاه علامت دار (**I64 (Signed 8 Bytes)**)
- 64 بیت یا 8 بایت، عدد صحیح با نگاه بدون علامت (**U64 (Unsigned 8 Bytes)**)
- 32 بیت یا 4 بایت، عدد اعشاری با نگاه علامت دار (**F32 (Float 4 Bytes)**)
- 64 بیت، یا 8 بایت، عدد اعشاری با نگاه علامت دار (**F64 (Float 8 Bytes)**)

شروع اولین اسکن

حال که با جزئیات اسکن آشنا شدیم، می‌توانیم اولین اسکن خود را شروع کنیم.

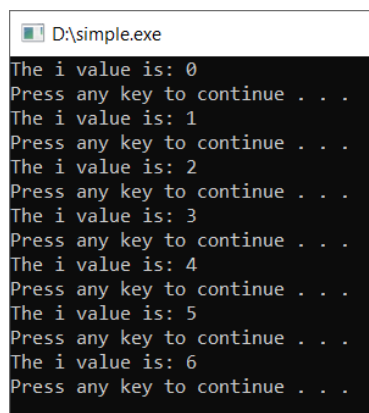
در ابتدا با یک مثال ساده شروع می‌کنیم.

کد زیر را که با زبان C نوشته شده است را در نظر بگیرید:

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

int main(){
    for(int i = 0;; i++){
        printf("The i value is: %d\n", i);
        system("pause");
    }
}
```

بصورت ساده، این برنامه هر بار مقدار کنونی متغیر **i** با مقدار اولیه صفر را در خروجی برنامه چاپ می‌کند و سپس برنامه متوقف می‌شود و منتظر می‌ماند که کاربر با فشار دادن دکمه ای به برنامه بگوید که ادامه دهد. و در ادامه، مقدار متغیر **i** به اندازه یک واحد زیاد می‌شود و مراحل قبل دوباره تکرار می‌شوند.

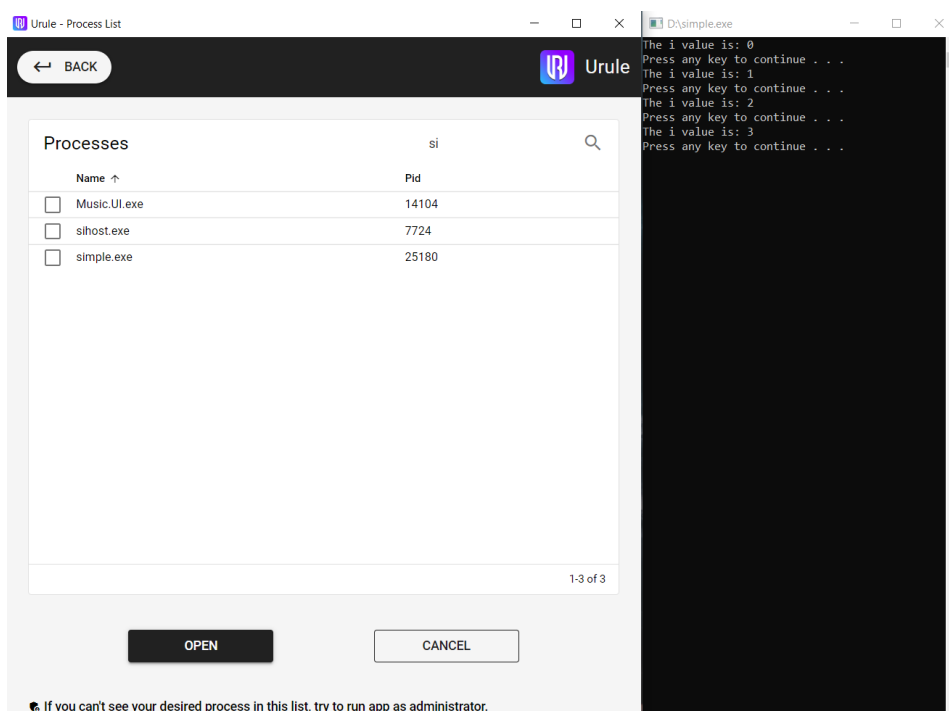


```
D:\simple.exe
The i value is: 0
Press any key to continue . . .
The i value is: 1
Press any key to continue . . .
The i value is: 2
Press any key to continue . . .
The i value is: 3
Press any key to continue . . .
The i value is: 4
Press any key to continue . . .
The i value is: 5
Press any key to continue . . .
The i value is: 6
Press any key to continue . . .
```

نمایی از اجرای برنامه

حال فرض کنید بخواهیم بدون اینکه کد برنامه را عوض کنیم کاری کنیم که مقدار متغیر i ، برابر با 1000- بشود، رسیدن به این هدف به این صورتی که برنامه نوشته شده کاری ناممکن یا حداقل سخت است. در ادامه تلاش می‌کنیم با استفاده از ابزاری که نوشتیم، این کار را به راحتی انجام دهیم.

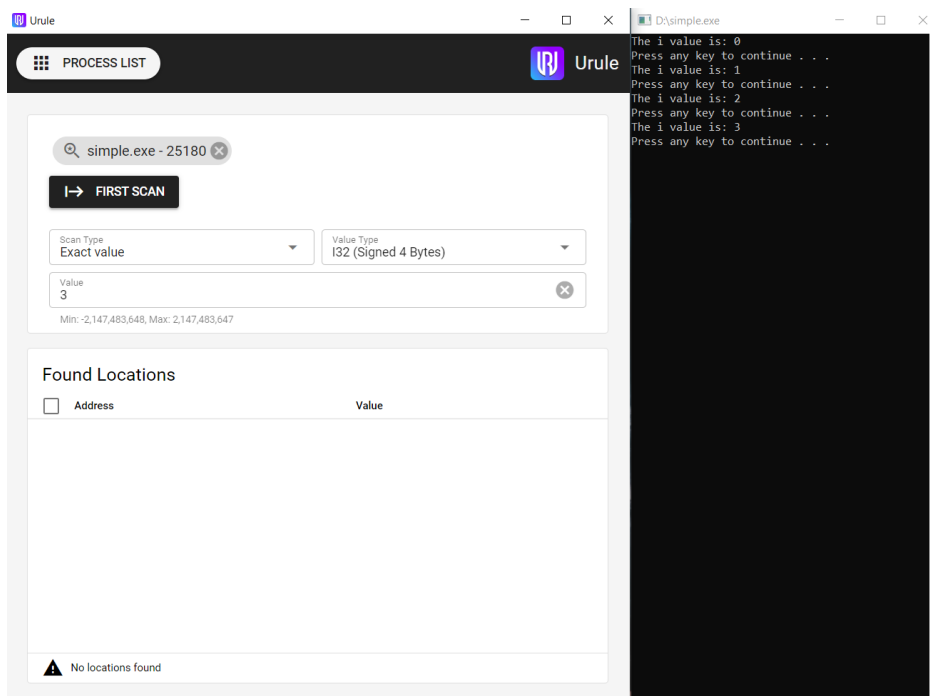
برای شروع، ابزار اسکن کننده و برنامه C نوشته شده مان را باز کرده و از لیست پروسه های در حال اجرا در سیستم، اسم پروسه برنامه نوشته شده مان را (در اینجا simple.exe) پیدا و آن را انتخاب می‌کنیم:



انتخاب پروسه از لیست پروسه های در حال اجرای سیستم

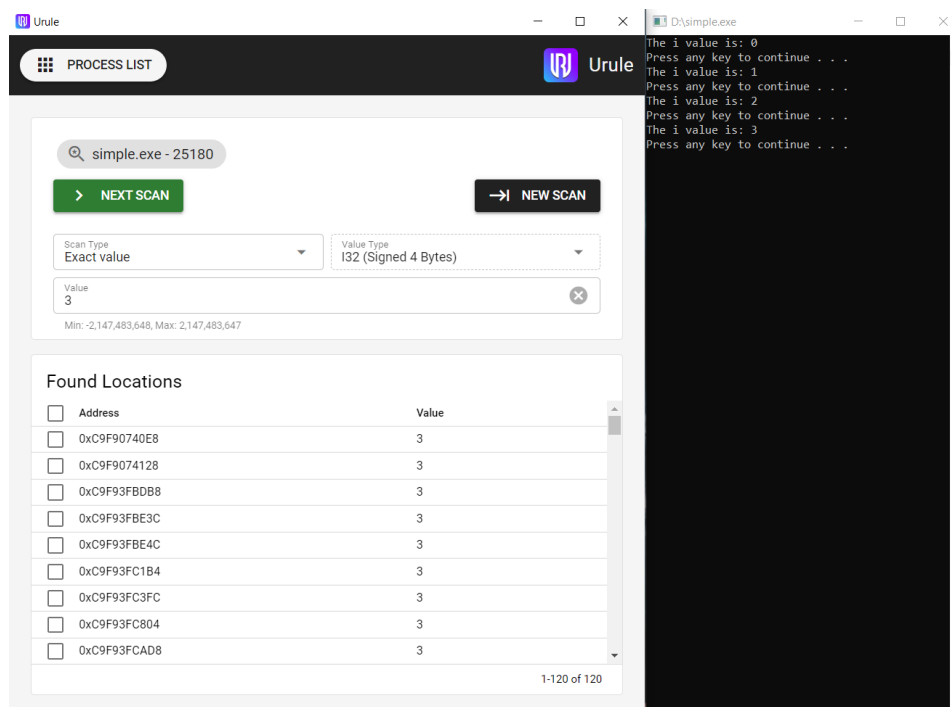
✓ در این قدم به این نکته توجه کنید که اگر نام پروسه ای که به دنبال آن بودید را در این لیست ندیدید، سعی کنید که برنامه را در حالت **Administrator** اجرا کنید.

حال در ادامه، نوع اسکن و تایپی مقداری که می خواهیم آن را جست و جو کنیم را انتخاب می کنیم. از آن جایی که مقدار دقیق متغیر **i** را در هر لحظه می دانیم، **اسکن مقدار دقیق (Exact value)** را انتخاب می کنیم و از آن جایی که در داخل کد نوشته شده می دانیم تایپ متغیر **i** از نوع **int** که همان **عدد صحیح 32 بیتی علامت دار** است و معمولاً هم اکثر متغیرهای عددی از این نوع هستند، تایپ مقداری (**I32 (Signed Bytes)**) را انتخاب می کنیم و در نهایت مقداری که به دنبال آن هستیم را وارد می کنیم:



وضعیت برنامه قبل از انجام اولین اسکن

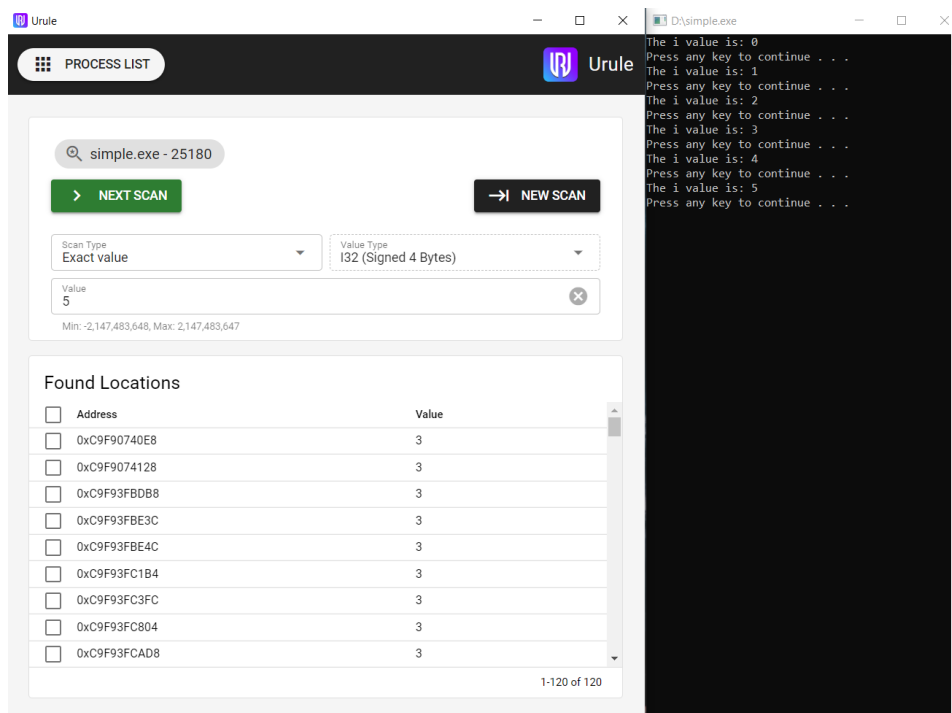
حال اولین اسکن رو شروع می‌کنیم و منتظر می‌مانیم که برنامه تمام خانه های حافظه ای که مقدارشان برابر 3 هست را برای ما پیدا کند:



وضعیت برنامه بعد از انجام اولین اسکن

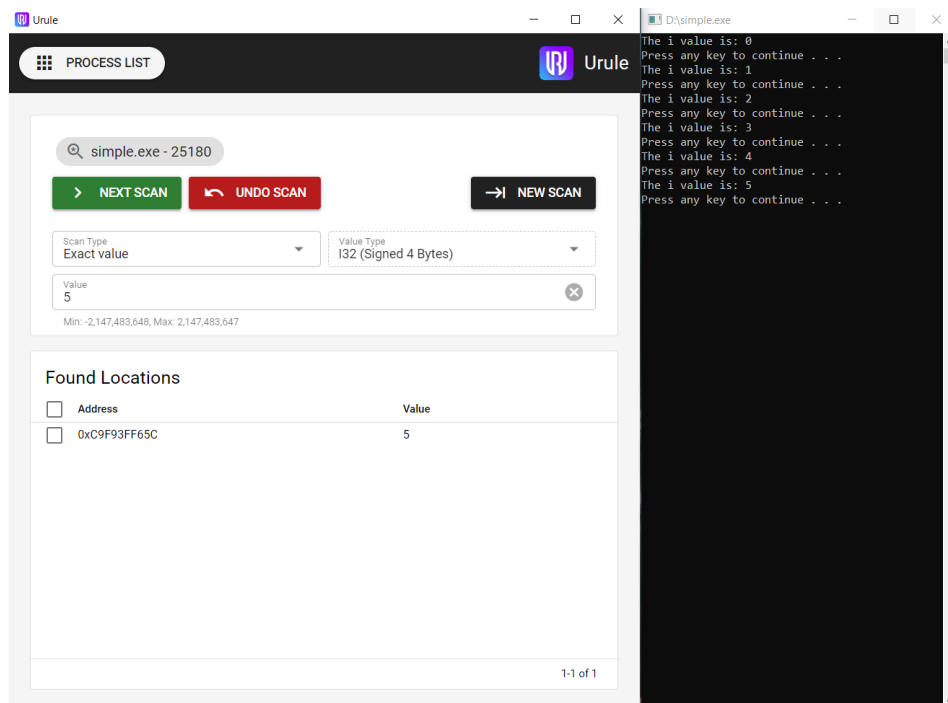
همین طور که در جدول پایین برنامه می‌بینیم، تعداد 120 عدد خانه حافظه با دید I32 برای ما پیدا شدند که مقدارشان برابر 3 است. ولی تعداد این حافظه های پیدا شده زیاد است، در صورتی که ما می‌دانیم متغیر i تنها صاحب یکی از این حافظه ها است و عوض کردن تمام این حافظه ها هم ممکن است باعث بشود که اجرای برنامه به مشکل بربخورد.

بدین منظور سعی می‌کنیم مقدار متغیر i را عوض کنیم و دوباره از بین مقادیر پیدا شده جست و جوی را ادامه دهیم:



وضعیت برنامه قبل از انجام دومین اسکن

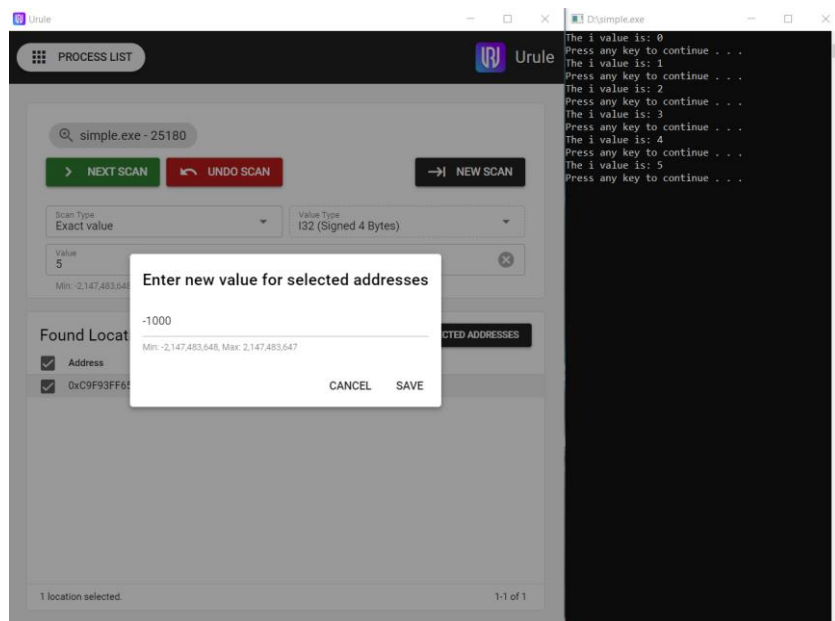
در اینجا مقدار متغیر **i** را از **3** به **5** تغییر دادیم، حال جست و جوی را ادامه می‌دهیم که ببینیم کدام یک از حافظه‌های پیدا شده حال مقدارشان برابر با 5 شده است:



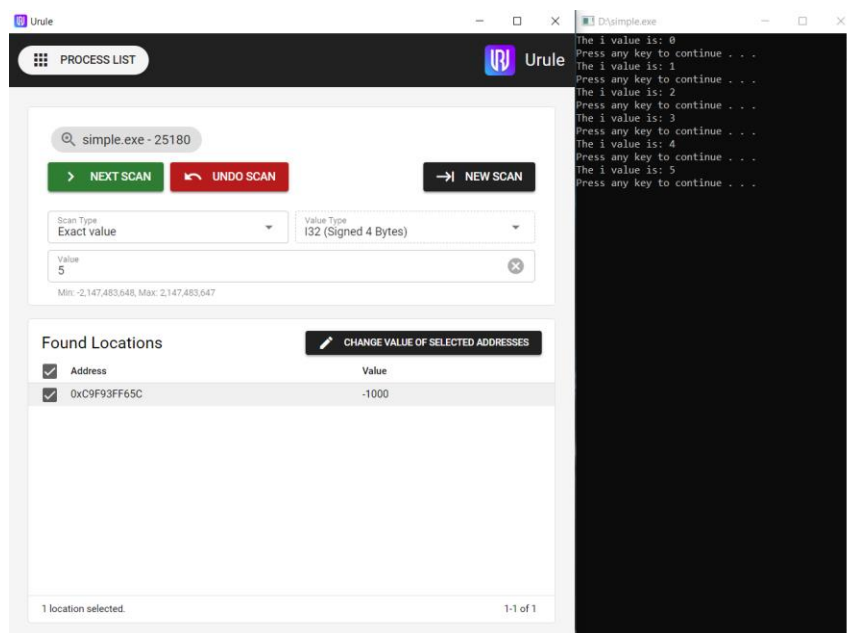
وضعیت برنامه بعد از انجام دومین اسکن

در اینجا می‌بینیم که تنها یک حافظه است که مقدارش همگام با متغیر *i* عوض شده است و این دقیقا همان خانه ای از حافظه است که به دنبال آن می‌گردیم!

حال که حافظه مربوط به متغیر *i* را پیدا کردیم، آن را انتخاب و مقدارش را به مقدار دلخواه خود عوض می‌کنیم:



در حال عوض کردن مقدار متغیر *i* به *-1000*



بعد از عوض کردن مقدار متغیر *i*

در این مرحله مقدار حافظه متغیر **i** برابر **1000**- شده است و می‌توانیم این را از جدول پایین برنامه ببینیم، حال برای اینکه از این معقوله مطمئن شویم برنامه نوشته شده را هم ادامه می‌دهیم تا نتیجه را در آن جا هم ببینیم:

The screenshot shows the Urule memory scanner interface on the left and a terminal window on the right. The scanner is set to search for the value 5 in the process simple.exe (PID 25180). It has found one location at address 0xC9F93FF65C with a value of -1000. The terminal window shows the output of a program where the variable 'i' is being incremented from 0 to 5, and then the program continues to print values from -999 to -992.

Urule Interface:

- Process: simple.exe - 25180
- Buttons: NEXT SCAN, UNDO SCAN, NEW SCAN
- Scan Type: Exact value
- Value Type: I32 (Signed 4 Bytes)
- Value: 5
- Min: -2,147,483,648, Max: 2,147,483,647
- Found Locations:

Address	Value
0xC9F93FF65C	-1000
- 1 location selected. 1-1 of 1

Terminal Output (D:\simple.exe):

```
The i value is: 0
Press any key to continue . . .
The i value is: 1
Press any key to continue . . .
The i value is: 2
Press any key to continue . . .
The i value is: 3
Press any key to continue . . .
The i value is: 4
Press any key to continue . . .
The i value is: 5
Press any key to continue . . .
The i value is: -999
Press any key to continue . . .
The i value is: -998
Press any key to continue . . .
The i value is: -997
Press any key to continue . . .
The i value is: -996
Press any key to continue . . .
The i value is: -995
Press any key to continue . . .
The i value is: -994
Press any key to continue . . .
The i value is: -993
Press any key to continue . . .
The i value is: -992
Press any key to continue . . .
```

در اینجا می‌بینیم که مقدار **i** واقعا عوض شده است

و تمام!

ادامه دادن اسکن ها

تا اینجا توانستیم مقدار های حافظه یک برنامه ساده را اسکن کنیم، از آن بخوانیم و در آن بنویسیم. اما همه برنامه هایی که ما در کامپیوتر خود اجرا می کنیم به این سادگی نیستند و حتی ممکن است کسانی دیگر جز ما آن را نوشته باشد و ما از جزئیات آن بی خبر باشیم.

مثلا ممکن است ندانیم تایپ متغیری که دنبال آن می گردیم چیست و یا حتی مقدار دقیق لحظه ای متغیری که دنبال آن می گردیم را نبینیم و فقط تغییرات یا بازه مقادیر را بدانیم.

ابزاری که در این پروژه نوشته شده نیز محدود به برنامه های ساده و انواع اسکن و تایپ های مقداری ساده نیست، بلکه همان طور که قبل گفته شد از انواع اسکن ها و تایپ های مقداری متفاوت و پیچیده تری نیز پشتیبانی می کند.

در ادامه، مثال های مختلفی از چند برنامه پیچیده تر و در مقیاس تجاری آورده شده اند.

مثال اول را با بازی [Stronghold Crusader](#) شروع می‌کنیم و مقدار پولی که بازی برای ما معین کرده را به صورت نامحدودی بیشتر می‌کنیم:

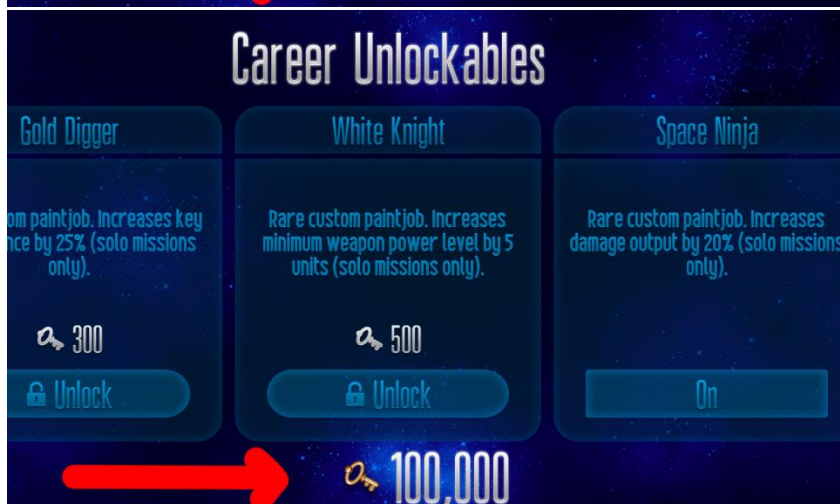
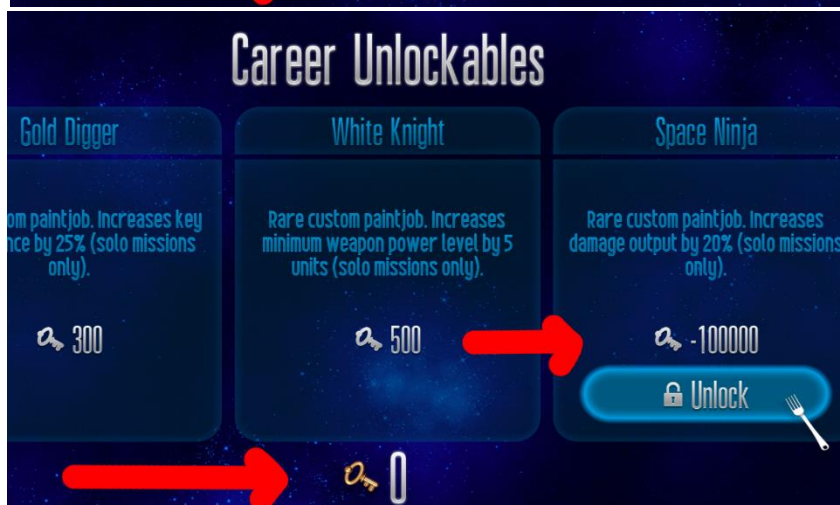


قبل از عوض کردن پول



بعد از عوض کردن پول

برای مثال دوم نیز سراغ بازی [Chicken Invaders](#) می‌رویم و در آن جا با تبدیل مقدار کلید خواسته شده برای باز کردن یک ویژگی به یک عدد منفی، نه تنها آن ویژگی را دریافت می‌کنیم و بلکه مقدار موجودی کلید خود را نیز زیاد می‌کنیم:



جزئیات پیاده سازی

برای پیاده سازی این پروژه از فریم ورک [Tauri](#) استفاده شده که به ما امکان ساخت یک برنامه بهینه، ایمن و مستقل از پلتفرم را می‌دهد.

ساختار یک برنامه ساخته شده با Tauri به صورت کلی به دو قسمت کلی: **هسته و رابط کاربری** تقسیم می‌گردد.

خود فریم ورک Tauri و هسته برنامه هایی که با آن نوشته می شوند با زبان برنامه نویسی [Rust](#) هستند که یک زبان مدرن و سطح پایین و با کارایی بالاست که شعارش "زبانی که همه را توانمند می کند که نرم افزار قابل اعتماد و کارآمد بنویسند" است و در سال های اخیر رقیبی برای زبان هایی مثل C و C++ شناخته می‌شود.

برای قسمت رابط کاربری نیز، Tauri دست ما را بسیار باز گذاشته و برای پیاده سازی رابط کاربری می توانیم از تمام فریم ورک های حوزه وب برای تولید رابط کاربری مانند: [Vuejs](#) ، [Reactjs](#) و ... استفاده کنیم.

برای قسمت رابط کاربری این پروژه نیز از فریم ورک Vuejs در کنار [Quasar](#) استفاده شده است که با داشتن اجزاء (component) از پیش آماده برای رابط کاربری، لازم نباشد تمامی اجزا را خودمان از پایه بنویسیم.

ویژگی های کلی برنامه

- ✓ استفاده بهینه از پردازنده و سرعت بالای اجرا، به علت اینکه هسته اجرایی برنامه با زبان Rust نوشته شده و فایل اجرایی این برنامه مستقیماً بدون واسطه توسط سیستم عامل اجرا می‌شوند.
- ✓ استفاده بهینه از حافظه، به دلیل آن که هسته اجرایی برنامه از گاریج کالکتور (GC) استفاده نمی‌کند و رابط کاربری نیز برای نشان دادن داده های بزرگ از صفحه بندی (pagination) استفاده می‌کند.
- ✓ استفاده آسان با رابط کاربری ساده و زیبا
- ✓ سبک برنامه (تقریباً 9 مگابایت)
- ✓ استفاده کردن از لاگ انداز (logger) برای بررسی مشکلات پیش آمده احتمالی
- ✓ استفاده از مفاهیم برنامه نویسی فانکشنال ([Functional programming](#)) در پیاده سازی
- ✓ استفاده حداکثر از حافظه استک (stack) و استفاده حداقل حافظه هیپ (heap) به منظور افزایش سرعت اجرای برنامه و کاهش استفاده از منابع سیستم
- ✓ استفاده از 6 روش مختلف ذخیره سازی آدرس های پیدا شده در هر اسکن، به منظور به حداقل رساندن استفاده از حافظه در حالات مختلف
- و در آخر پروژه به صورت متن باز (open source) توسعه داده شده است و تمامی کدهای منبع آن در مخزن (repository) زیر قابل مشاهده هستند:

<https://github.com/mohsenpakzad/urule>