قسمت ۷ – تجزیه و تحلیل کاربردی بدافزارها

راهنمای جامع مهندسی معکوس، تجزیه و تحلیل بدافزارها،
باج‌افزارها، جاسوس‌افزارها، روت‌کیت‌ها و بوتکیت‌های کامپیوترای

آزمایشگاه امنیت کی‌پاد

نویسنده: میلاد کهساری الهادی

homeland security
kaipod

کیان امنیت
کیپاد

شناخت ساختمان‌های کد C در دیزاسمبلی[1]

در قسمت‌های گذشته این سلسله مقالات، معماری x86 و برخی از دستورالعمل‌های رایج این معماری بررسی شد اما افرادی که مهندسی معکوس یک باینری را با موفقیت انجام می‌دهند، هر یک از دستورالعمل‌ها را به صورت مجزا ارزیابی نمی‌کنند، مگر این‌که به این کار نیاز داشته باشند. زیرا این فرایند بسیار خسته‌کننده است و ممکن است تعداد دستورالعمل‌ها برای یک برنامه دیزاسمبل شده به هزاران یا میلیون‌ها دستورالعمل اسمبلی برسد. به عنوان یک تحلیلگر بدافزار، باید بتوانید با تحلیل یک گروه از دستورالعمل‌های اسمبلی یک تصویر سطح بالا از عملکرد برنامه به‌دست آورید و در مواقع مورد نیاز روی تحلیل هر دستورالعمل بطور مجزا متمرکز شوید. با این حال، به‌دست آوردن این مهارت به زمان بسیاری نیاز دارد.

در این قسمت در مورد چگونگی توسعه کد توسط نویسندگان بدافزار و تبدیل آن‌ها به یک گروه از دستورالعمل‌های اسمبلی بحث می‌کنیم. بدافزارها معمولاً با یک زبان سطح بالا همچون زبان C که رایج‌ترین انتخاب نویسندگان بدافزار می‌باشد، توسعه داده می‌شوند. ساختمان کد[2] یک لایه انتزاعی برای کد است که مشخصه کارایی یک برنامه را تعریف می‌کند، اما جزییات اجرای[3] آن را ارائه نمی‌دهد. نمونه‌هایی از ساختمان کدهای سطح بالا شامل حلقه‌ها[4]، عبارات شرطی IF/Else، لیست‌های پیوندی[5]، عبارات سوئیچ[6] و غیره ... هستند. شایان ذکر است، برنامه‌ها می‌توانند به ساختمان‌های مجزا تقسیم شوند و سپس به منظور پیاده‌سازی عملکرد کلی برنامه با همدیگر ترکیب شوند.

این قسمت با بیش از ۱۰ مثال از بلاک کدهای اصلی زبان C طراحی شده است تا شما را در مسیر خود هدایت کند. در این قسمت هر بلاک کد C در اسمبلی بررسی خواهد شد، هدف از این قسمت این است که به شما در انجام عملیات مهندسی‌معکوس یک باینری کمک کند و هدف شما به عنوان یک تحلیلگر بدافزار باید رفتن از سطح دیزاسمبلی به یک ساختمان سطح بالا باشد. زیرا برنامه‌نویسان به خواندن و درک کد منبع عادت کرده‌اند.

---

[1] Recognizing C code Constructs in Assembly
[2] Code Construct
[3] Implementation
[4] loops
[5] Linked lists
[6] Switch

homeland security kaipod
کـــیان امنیت کایپد

این قسمت روی چگونگی کامپایل شدن بلاک‌های رایج از قبیل حلقه‌ها و عبارات شرطی متمرکز خواهد شد. پس از آشنایی شما با این مفاهیم، خواهید آموخت چگونه یک شکل سطح بالا از کارایی یک بدافزار به سرعت به‌دست آورید. علاوه بر این به بحث درباره ساختمان داده‌های مختلف، همچنین به بررسی تفاوت‌های میان کامپایلرها خواهیم پرداخت، زیرا نسخه‌ها و تنظیمات مختلف کامپایلرها می‌توانند در چگونگی نمایش یک ساختمان خاص در محیط دیزاسمبلی تاثیر بگذارند. در این قسمت، ما به دو روش مختلف عبارات سوئیچ و فراخوانی‌های توابع را که با استفاده از کامپایلرهای مختلف کامپایل شده‌اند، بررسی خواهیم کرد. این فصل عمیقا ساختار کدهای C را بررسی خواهد کرد، بنابراین در این قسمت از کتاب درباره زبان C و در حالت کلی برنامه‌نویسی درک بیشتری دریافت خواهید کرد.

اکثریت بدافزارها با زبان C نوشته شده‌اند، اما ممکن است برخی از آن‌ها با زبان‌های دیگر همچون دلفی یا C++ نوشته شوند. در هر حال، زبان C یک زبان برنامه‌نویسی ساده با یک رابطه جداناپذیر با اسمبلی است. بنابراین اینجا، منطقی‌ترین مکان برای تحلیلگران بدافزار تازه کار به منظور شروع است. به یاد داشته باشید، هدف شما درک عملکرد کلی برنامه است و نباید تک تک دستورالعمل‌های یک برنامه را مورد تحلیل قرار بدهید. این را به ذهن بسپارید، روی نحوه عملکرد برنامه در حالت کلی متمرکز شوید و چگونگی عملکرد آن برای هر چیز خاص را مورد بررسی قرار ندهید. شایان ذکر است، در قسمت‌های بعدی این سلسله مقالات، درباره تحلیل کدهای C++ همچنین بحث خواهیم کرد.

متغیرهای گلوبال در مقابل لوکال[1]

متغیرهای گلوبال می‌توانند توسط هر تابع درون برنامه در دسترس و مورد استفاده قرار گیرند. متغیرهای لوکال توسط توابعی فقط می‌توانند مورد استفاده قرار گیرند که در آن تعریف شده‌اند. شایان ذکر است، متغیرهای لوکال و متغیرهای گلوبال به صورت مشابه در C تعریف می‌شوند، اما ظاهر آن‌ها در دیزاسمبلی کاملا متفاوت است.

[1] Global vs. Local Variables

در زیر دو نمونه کد C برای متغیرهای گلوبال و متغیرهای لوکال آورده شده است. به تفاوت میان آن دو توجه کنید. مثال متغیرهای گلوبال لیست ۱ است که x و y را به عنوان متغیرهای گلوبال خارج از تابع تعریف کرده‌اند. مثال متغیرهای لوکال لیست ۲ است که در آن متغیرهای درون تابع تعریف شده‌اند.

```
1  #include <windows.h>
2  #include <iostream>
3
4  int x = 1;
5  int y = 2;
6
7  void main()
8  {
9      x = x + y;
10     printf("Total = %d\n", x);
11 }
```

لیست ۱: یک برنامه ساده با دو متغیر گلوبال

```
1  #include <windows.h>
2  #include <iostream>
3
4  void main()
5  {
6      int x = 1;
7      int y = 2;
8      x = x + y;
9      printf("Total = %d\n", x);
10 }
```

لیست ۲: یک برنامه ساده با دو متغیر لوکال

تفاوت میان متغیرهای لوکال و متغیرهای گلوبال در ساختار زبان C بسیار ناچیز است و در این حالت نتیجه برنامه مشابه می‌باشد. اما در حالت دیزاسمبل شده که خروجی آن در لیست ۳ و لیست ۴ نمایش داده شده است، تفاوت‌ها بسیاری بدیهی است. متغیرهای گلوبال با آدرس‌های حافظه و متغیرهای لوکال با آفستی نسبت به آدرس پشته ارجاع داده شده‌اند.

در لیست ۳، متغیر گلوبال x توسط یک آدرس حافظه در 0x40CF60 مشخص شده است. همچنین به این نکته توجه کنید، هنگامی‌که مقدار dword_40CF60 به درون ثبات eax منتقل می‌شود و عملیات ADD

یا جمع صورت می‌گیرد، در ادامه متغیر x (شماره ۱) با یک مقدار جدید تعویض خواهد شد. سپس تمامی توابع بعدی که این متغیر را به کار گرفتند، تحت تاثیر این جایگزینی قرار خواهند گرفت.

```
00401003        mov     eax, dword_40CF60
00401008        add     eax, dword_40C000
0040100E        mov     dword_40CF60, eax ❶
00401013        mov     ecx, dword_40CF60
00401019        push    ecx
0040101A        push    offset aTotalD  ;"total = %d\n"
0040101F        call    printf
```

لیست ۳: کد اسمبلی برای مثال متغیر گلوبال در لیست ۱ است.

در لیست‌های ۴ و ۵، متغیر محلی x در محلی روی پشته نسبت به آفست ثابتی از ثبات EBP واقع شده است. در لیست ۴، محل حافظه [ebp-4] بطور مداوم در سراسر این تابع برای اشاره به متغیر محلی x استفاده شده است. این دستورالعمل به ما می‌گوید که ebp-4 یک متغیر محلی مبتنی بر پشته است که فقط در تابعی که در آن تعریف شده است، مورد ارجاع قرار می‌گیرد.

```
00401006        mov     dword ptr [ebp-4], 0
0040100D        mov     dword ptr [ebp-8], 1
00401014        mov     eax, [ebp-4]
00401017        add     eax, [ebp-8]
0040101A        mov     [ebp-4], eax
0040101D        mov     ecx, [ebp-4]
00401020        push    ecx
00401021        push    offset aTotalD  ; "total = %d\n"
00401026        call    printf
```

لیست 4: کد اسمبلی برای مثال متغیر محلی در لیست ۲، بدون برچسب‌گذاری است.

در لیست ۵، x به صورت مطلوب توسط اسمبلر IDA Pro با نام گنگ var_4 برچسب‌گذاری شده است. همان‌طور که در قسمت قبل بحث کردیم، اسامی گنگ را می‌توان نام‌گذاری مجدد کرد تا با مفهوم شوند و کارکرد خود را انعکاس دهند. داشتن این متغیر لوکال با نام var_4 بجای 4- کار تحلیل ما را ساده‌تر می‌کند، چون هنگامی‌که شما var_4 را به x تغییر نام می‌دهید، دیگر در سراسر تابع نیاز به پیگیری آفست ۴- نخواهید داشت.

```
00401006        mov     [ebp+var_4], 0
0040100D        mov     [ebp+var_8], 1
00401014        mov     eax, [ebp+var_4]
00401017        add     eax, [ebp+var_8]
0040101A        mov     [ebp+var_4], eax
0040101D        mov     ecx, [ebp+var_4]
00401020        push    ecx
00401021        push    offset aTotalD  ; "total = %d\n"
00401026        call    printf
```

لیست ۵: کد اسمبلی برای متغیر لوکال لیست ۲ با برچسب‌گذاری

دیزاسمبل عملیات‌های محاسباتی[1]

بسیاری از انواع مختلف عملیات‌های ریاضی می‌توانند در حین برنامه‌نویسی C انجام گیرند. در این قسمت ما این عملیات‌ها را دیزاسمبل کرده و به شما ارائه خواهیم داد. لیست ۶ یک کد C را که شامل دو متغیر و انواع مختلف عملیات‌های محاسباتی می‌شود، نمایش می‌دهد. دو تا از عملیات‌های محاسباتی، عملیات‌های افزایشی ++ و کاهشی -- هستند که مقدار موجود در یک متغیر را یک واحد افزایش یا کاهش می‌دهند. عملوند ٪ هم دو متغیر را با هم تقسیم می‌کند و سپس در خروجی باقی مانده را صادر می‌نماید.

```
1   // Code snippet by Milad Kahsari Alhadi
2
3   #include <windows.h>
4   #include <iostream>
5
6   int main(int argc, const char* argv[])
7   {
8       int a = 0;
9       int b = 1;
10
11      a = a + 11;
12      a = a - b;
13
14      a--;
15      b++;
16
17      b = a % 3;
18
19      return 0;
20  }
```

لیست ٦: یک برنامه با زبان C دارای دو متغیر و عملیات‌های محاسباتی مختلف

[1] Disassembling Arithmetic Operations

لیست ۷ نمایش دیزاسمبلی کد C آورده شده در لیست ۶ را ارائه می‌دهد، که می‌توانید دوباره به زبان C ترجمه شود. برای دریافت یک نمایش سطح بالا از این دستورات دیزاسمبلی، کافی است معادل نمایش آن در زبان C را به خاطر بسپارید.

```
00401006    mov     [ebp+var_4], 0
0040100D    mov     [ebp+var_8], 1
00401014    mov     eax, [ebp+var_4] ❶
00401017    add     eax, 0Bh
0040101A    mov     [ebp+var_4], eax
0040101D    mov     ecx, [ebp+var_4]
00401020    sub     ecx, [ebp+var_8] ❷
00401023    mov     [ebp+var_4], ecx
00401026    mov     edx, [ebp+var_4]
00401029    sub     edx, 1 ❸
0040102C    mov     [ebp+var_4], edx
0040102F    mov     eax, [ebp+var_8]
00401032    add     eax, 1 ❹
00401035    mov     [ebp+var_8], eax
00401038    mov     eax, [ebp+var_4]
0040103B    cdq
0040103C    mov     ecx, 3
00401041    idiv    ecx
00401043    mov     [ebp+var_8], edx ❺
```

لیست ۷: کد دیزاسمبلی برای عملیات‌های محاسباتی لیست ٦

در این مثال، a و b متغیرهای محلی هستند، زیرا آن‌ها توسط پشته ارجاع داده شده‌اند. دیزاسمبلر IDA Pro متغیر a را به عنوان var_4 و متغیر b را به عنوان var_8 برچسب‌گذاری کرده است. سپس var_4 و var_8 با مقادیر ۰ و ۱ به ترتیب مقداردهی شده‌اند.

سپس a به درون ثبات eax منتقل شده (شماره ۱) و سپس مقدار 0x0b به آن افزوده شده است. در نتیجه این عملیات متغیر a به مقدار ۱۱ واحد افزایش پیدا کرده است و سپس متغیر b از متغیر a (شماره ۲) کم شده است. (در این مثال، کامپایلر تصمیم به استفاده از دستورالعمل‌های add و sub (شماره ۳ و ۴) بجای استفاده از دستورالعمل‌های inc و dec گرفته است).

پنج دستورالعمل اسمبلی نهایی عملیات محاسبه باقی مانده را پیاده‌سازی می‌کنند. هنگام اجرای دستورالعمل‌های div یا idiv (شماره ۵)، ثبات‌های edx:eax با عملوند دستورالعمل‌های div یا idiv تقسیم شده و نتیجه عملیات در ثبات Eax و باقی مانده عملیات تقسیم در ثبات edx ذخیره می‌شود. به این دلیل است که مقدار ثبات edx به درون var_8 (شماره ۵) منتقل می‌شود.

شناخت عبارات شرطی

برنامه‌نویسان از دستورات شرطی برای تعویض مسیر اجرایی معمول برنامه بر مبنای برخی شروط خاص استفاده می‌کنند. عبارات و دستورالعمل‌های شرطی در زبان برنامه‌نویسی C و دیزاسمبلی بسیار رایج هستند. در این قسمت ما عبارت‌های شرطی ساده و سپس عبارت‌های شرطی تو در تو را بررسی خواهیم کرد. هدف شما در این قسمت باید یادگیری چگونگی شناسایی انواع مختلف عبارات شرطی باشد.

لیست ۸ یک عبارت شرطی ساده if در زبان برنامه‌نویسی C و لیست ۹ کد دیزاسمبل شده این عبارت شرطی را نمایش می‌دهند. با توجه به دستورالعمل پرش شرطی jnz (شماره ۲)، باید یک پرش شرطی مبتنی بر عبارت if وجود داشته باشد، با این حال همه پرش‌های شرطی مطابق با عبارت if نیستند.

```
1   // Code snippet by Milad Kahsari Alhadi
2
3   #include <windows.h>
4   #include <iostream>
5
6   int main(int argc, const char* argv[])
7   {
8       int x = 1;
9       int y = 2;
10
11      if (x == y) {
12          printf("x equals y.\n");
13      }
14      else {
15          printf("x is not equal to y.\n");
16      }
17
18      return 0;
19  }
```

لیست ۸: مثال عبارت شرطی **if** در زبان **C**

```
00401006         mov     [ebp+var_8], 1
0040100D         mov     [ebp+var_4], 2
00401014         mov     eax, [ebp+var_8]
00401017         cmp     eax, [ebp+var_4] ❶
0040101A         jnz     short loc_40102B ❷
0040101C         push    offset aXEqualsY_ ; "x equals y.\n"
00401021         call    printf
00401026         add     esp, 4
00401029         jmp     short loc_401038 ❸
0040102B loc_40102B:
0040102B         push    offset aXIsNotEqualToY ; "x is not equal to y.\n"
00401030         call    printf
```

لیست ۹: کد دیزاسمبلی عبارت شرطی **if** لیست ۸

همان‌طور که در لیست ۹ مشاهده می‌کنید، قبل از اجرای عبارت شرطی درون کد لیست ۹ باید تصمیمی گرفته شود. این شرط مطابق با پرش شرطی نمایش داده شده در شماره ۲ خروجی لیست ۸ است. شایان ذکر است، تصمیم پرش بر مبنای مقایسه cmp صورت می‌گیرد، که بررسی می‌کند آیا متغیرهای var_4 و var_8 با هم دیگر برابر هستند یا خیر (متغیرهای var_4 و var_8 مطابق متغیرهای x و y در کد منبع هستند).

با این حال اگر این دو متغیر با هم برابر نباشند، در نتیجه پرش رخ می‌دهد و برنامه در خروجی پیام " x is not equal to y" به معنا این‌که متغیر x با y برابر نیست، چاپ می‌کند؛ در غیر این صورت برنامه مسیر اجرای خود را ادامه می‌دهد و در پایان پیام "x equals y" را به معنا این‌که متغیر x برابر با y است، به خروجی صادر خواهد کرد. همچنین به دستورالعمل پرش (JMP) (شماره ۳) توجه کنید که می‌تواند مسیر اجرای برنامه را به هر نقطه از کد برنامه منتقل کند.

تجزیه و تحلیل گرافیکی توابع با IDA Pro

دیزاسمبلر IDA Pro یک ابزار تولید گراف دارد که در تشخیص ساختمان توابع بسیار مفید می‌باشد، این ویژگی در تصویر ۱ نمایش داده شده است. شایان ذکر است این ویژگی نمایش پیش‌فرض IDA Pro برای تحلیل توابع است. تصویر ۱ گرافی از کد دیزاسمبلی برای مثال لیست ۹ را نمایش می‌دهد. همان‌طور که مشاهده می‌کنید، دو راه مختلف (شماره ۱ و شماره ۲) وجود دارد که منجرب به پایان تابع می‌شوند و هر کدام از آن‌ها یک رشته متفاوت را در خروجی چاپ می‌کنند.

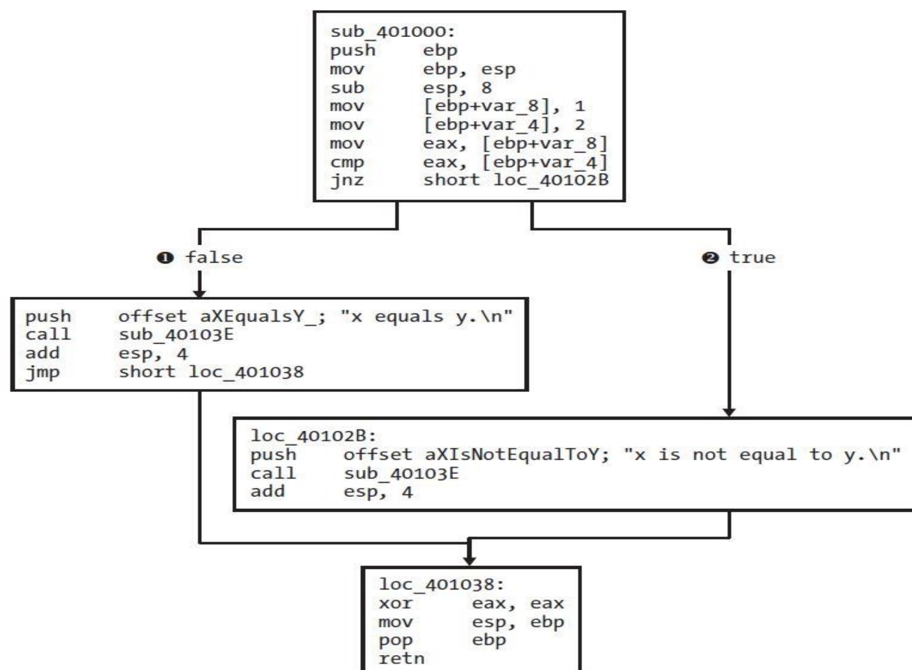
مسیر شماره ۱ پیام ".x equals y" را در خروجی چاپ خواهد کرد و مسیر دوم پیام "x is not equal to y." را در خروجی چاپ می‌کند. همچنین دیزاسمبلر IDA Pro برچسب‌های false و true را روی مسیرها برچسب‌گذاری کرده است که این برچسب‌ها مشخص می‌کنند اجرای برنامه بر مبنای درست بودن یا اشتباه بودن شرط از کدام مسیر عبور خواهد کرد. حال می‌توانید تصور کنید چه مقدار ویژگی تولید گراف توابع می‌تواند سرعت مهندسی‌معکوس را بالا ببرد.

شناخت عبارات if تو در تو

لیست ۱۰ یک کد به زبان C برای عبارات if تو در تو نمایش می‌دهد که شبیه به کد لیست ۸ است، با این تفاوت‌که دو عبارت if دیگر به درون عبارات if تو در تو افزوده شده است. این عبارات اضافی مشخص می‌سازند که آیا Z برابر با 0 است یا خیر.

```c
6   int main(int argc, const char* argv[])
7   {
8       int x = 0;
9       int y = 1;
10      int z = 2;
11
12      if (x == y) {
13          if (z == 0) {
14              printf("z is zero and x = y.\n");
15          }
16          else {
17              printf("z is non-zero and x = y.\n");
18          }
19      }
20      else {
21          if (z == 0) {
22              printf("z zero and x != y.\n");
23          }
24          else {
25              printf("z non-zero and x != y.\n");
26          }
27      }
28
29      return 0;
30  }
```

لیست ۱۰: عبارت **if** تو در تو



تصویر ۱: دیزاسمبلی گراف برای عبارت **if** مثال موجود در لیست ۹

همان‌طور که در لیست ۱۱ مشاهده می‌کنید، علیرغم آن تغییر جزیی ایجاده شده درون کد C، دیزاسمبلی آن خیلی پیچیده‌تر شده است.

```
00401006         mov    [ebp+var_8], 0
0040100D         mov    [ebp+var_4], 1
00401014         mov    [ebp+var_C], 2
0040101B         mov    eax, [ebp+var_8]
0040101E         cmp    eax, [ebp+var_4]
00401021         jnz    short loc_401047 ❶
00401023         cmp    [ebp+var_C], 0
00401027         jnz    short loc_401038 ❷
00401029         push   offset aZIsZeroAndXY_ ; "z is zero and x = y.\n"
0040102E         call   printf
00401033         add    esp, 4
00401036         jmp    short loc_401045
00401038 loc_401038:
00401038         push   offset aZIsNonZeroAndX ; "z is non-zero and x = y.\n"
0040103D         call   printf
00401042         add    esp, 4
00401045 loc_401045:
00401045         jmp    short loc_401069
00401047 loc_401047:
00401047         cmp    [ebp+var_C], 0
0040104B         jnz    short loc_40105C ❸
0040104D         push   offset aZZeroAndXY_ ; "z zero and x != y.\n"
00401052         call   printf
00401057         add    esp, 4
0040105A         jmp    short loc_401069
0040105C loc_40105C:
0040105C         push   offset aZNonZeroAndXY_ ; "z non-zero and x != y.\n"
00401061         call   printf00401061
```

لیست ۱۱: دیزاسمبلی عبارت if تو در تو لیست ۱۰

همان‌طور که در لیست ۱۱ مشاهده می‌کنید، سه پرش شرطی مختلف رخ داده است. اولین آن‌ها اگر var_4 با var_8 برابر نباشد، رخ می‌دهد (شماره ۱) و دوتای دیگر زمانی‌که var_C با ۰ برابر نباشد (شماره ۲ و ۳) رخ خواهد داد.

شناختن حلقه‌ها

حلقه‌ها و تکرارکننده وظایف در تمامی نرم‌افزارها بسیار رایج هستند و این بسیار مهم است که شما بتوانید آن‌ها را شناسایی کنید. در ادامه به تحلیل ساختار این نوع دستورالعمل‌ها در دیزاسمبلی خواهیم پرداخت.

شناسایی حلقه‌های For

دستورالعمل for یک مکانیزم ایجاد حلقه ساده و اساسی مورد استفاده در برنامه‌نویسی با زبان C است. حلقه‌های for همیشه چهار مولفه اصلی از قبیل مقداردهی اولیه[۱]، مقایسه‌کننده[۲]، دستورالعمل‌اجرایی[۳] و افزایش‌دهنده و کاهش‌دهنده عملوند شرط دارند. لیست ۱۲ یک مثال از حلقه for را نمایش می‌دهد.

```
1   // Code snippet by Milad Kahsari Alhadi
2
3   #include <windows.h>
4   #include <iostream>
5
6   int main(int argc, const char* argv[])
7   {
8       for (int i = 0; i < 100; i++) {
9           printf("i equals %d\n", i);
10      }
11
12      return 0;
13  }
```

لیست ۱۲: یک حلقه for در زبان C

همان‌طور که در مثال بالا مشاهده می‌کنید، متغیر i با مقدار عددی ۰ مقداردهی اولیه شده است، متغیر i اولین مولفه شرط حلقه دستورالعمل for می‌باشد. مولفه دوم، مقایسه‌کننده حلقه for است که بررسی می‌کند دستورالعمل درون حلقه یعنی printf تا زمانی‌که i کمتر از ۱۰۰ باشد، اجرا شود. سپس در هر بار اجرای حلقه یک مقدار با استناد به ++i به متغیر i افزوده می‌شود و پیوسته مقایسه کننده حلقه مقدار i را با ۱۰۰ بررسی می‌کند. این فرایند تا زمانی‌که i کم‌تر از مقدار ۱۰۰ باشد، ادامه پیدا خواهد کرد و هر گاه i برابر با ۱۰۰ شود اجرای برنامه از حلقه خارج می‌شود.

در اسمبلی، حلقه‌های تکرار for می‌تواند با مشاهده چهار مولفه مقداردهی اولیه، مقایسه کننده، دستورالعمل‌های اجرایی و افزایشی و کاهشی مورد شناسایی قرار گیرند. به عنوان مثال، در لیست ۱۳ (شماره ۱) مربوط به گام مقداردهی اولیه حلقه for است. کد میان (شماره ۳) و (شماره ۴) مربوط به عملوند افزایشی است و در قسمت (شماره ۲) دستورالعمل پرشی وجود دارد که اجرای برنامه را به نقطه loc_401016 منتقل می‌کند.

---

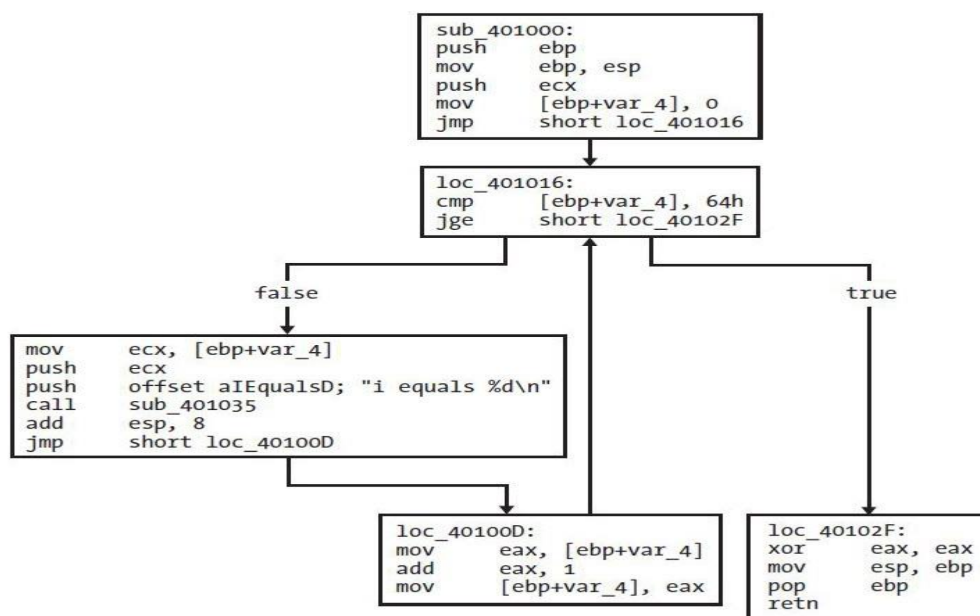
[1] Initialization
[2] Comparison
[3] Execution instructions

سپس عملیات مقایسه در در قسمت (شماره ۵) و (شماره ۶) رخ می‌دهد و تصمیم گیری بر مبنای مقایسه توسط یک پرش شرطی (jge = jump greater equal) صورت می‌گیرد. در پایان با استناد به شرط دستور، اگر پرش رخ ندهد، دستورالعمل printf اجرا خواهد شد و سپس یک پرش غیر شرطی در قسمت شماره ۷ رخ خواهد داد که موجب می‌شود اجرای برنامه به loc_40100D منتقل شود و در نتیجه عملوند شرط یک واحد افزایش پیدا کند.

```
00401004          mov     [ebp+var_4], 0 ❶
0040100B          jmp     short loc_401016 ❷
0040100D loc_40100D:
0040100D          mov     eax, [ebp+var_4] ❸
00401010          add     eax, 1
00401013          mov     [ebp+var_4], eax ❹
00401016 loc_401016:
00401016          cmp     [ebp+var_4], 64h ❺
0040101A          jge     short loc_40102F ❻
0040101C          mov     ecx, [ebp+var_4]
0040101F          push    ecx
00401020          push    offset aID  ; "i equals %d\n"
00401025          call    printf
0040102A          add     esp, 8
0040102D          jmp     short loc_40100D ❼
```

لیست ۱۳: کد اسمبلی مثال موجود در لیست ۱۲

همان‌طور که در تصویر ۲ نمایش داده شده است، حلقه‌های for می‌توانند با استفاده از حالت تولید گراف IDA Pro مورد شناسایی قرار گیرند.



تصویر ۲: گراف دیزاسمبلی برای حلقه for مثال موجود در لیست ۱۳

homeland security kaipod
کیان امنیت کایپد

در شکل بالا، فلش اشاره کننده به سمت بالا پس از کد افزایشی نشان‌دهنده یک حلقه می‌باشد. با این حال می‌توان گفت، این فلش‌ها در حالت گراف کار شناسایی حلقه‌ها را بسیار ساده‌تر از حالت دیزاسمبلی IDA Pro می‌کند. حالت گراف پنج جعبه نمایش می‌دهد: چهار جعبه بالایی مولفه‌های حلقه هستند (مقداردهی‌کننده اولیه، مقایسه‌کننده، قسمت اجرایی حلقه و قسمت افزایش دهنده مولفه شرط به ترتیب می‌شوند). جعبه پایینی موجود در سمت راست epilogue تابع است که ما در قسمت ۴ این مقالات آن را به عنوان پاسخگو به پاک‌سازی پشته و بازگشت به تابع اصلی آن را تشریح کردیم.

شناسایی حلقه‌های While

حلقه‌های تکرار While به صورت پیوسته توسط نویسندگان بدافزار برای ایجاد حلقه‌ای از دستورالعمل‌های اجرایی با دریافت یک شرط مورد استفاده قرار می‌گیرند. حلقه‌های While مشابه حلقه‌های for می‌باشند، اما آن‌ها بسیار ساده‌تر قابل شناسایی هستند. حلقه While در لیست ۱۴ بطور پیوسته به کار خود ادامه خواهد داد تا زمانی‌که checkResult مقدار ۰ را بازگشت دهد.

```
1  // Code snippet by Milad Kahsari Alhadi
2
3  #include <windows.h>
4  #include <iostream>
5
6  int performAction();
7  int checkResult(int arg_param);
8
9  int main(int argc, const char* argv[])
10 {
11     int status = 0;
12     int result = 0;
13
14     while (status == 0) {
15         result = performAction();
16         status = checkResult(result);
17     }
18
19     return 0;
20 }
```

لیست ١٤: یک حلقه **while** در زبان **C**

homeland security kaipod
کیاپاد کـــیـــان امنیت

کدهای اسمبلی در مثال ۱۵ به نظر می‌رسد شبیه به حلقه for است، به جزء این‌که فاقد یک بخش افزایشی می‌باشد. یک پرش شرطی در قسمت شماره ۱ و یک پرش غیر شرطی در شماره ۲ رخ داده است. اما با این حال تنها راه برای متوقف سازی اجرای تکراری این کد رخ دادن پرش شرطی است.

```
00401036        mov     [ebp+var_4], 0
0040103D        mov     [ebp+var_8], 0
00401044 loc_401044:
00401044        cmp     [ebp+var_4], 0
00401048        jnz     short loc_401063 ❶
0040104A        call    performAction
0040104F        mov     [ebp+var_8], eax
00401052        mov     eax, [ebp+var_8]
00401055        push    eax
00401056        call    checkResult
0040105B        add     esp, 4
0040105E        mov     [ebp+var_4], eax
00401061        jmp     short loc_401044 ❷
```

لیست ۱۵: کد اسمبلی برای حلقه while مثال موجود در لیست ۱٤

درک قراردادهای فراخوانی توابع

در قسمت چهارم این مقالات، ما درباره چگونگی استفاده از پشته و دستورالعمل call برای فراخوانی توابع بحث کردیم. فراخوانی توابع و قراردادهای حاکم که در راه فراخوانی توابع رخ می‌دهند، می‌توانند به صورت‌های مختلفی در کدهای دیزاسمبلی ظاهر شوند. این قراردادها شامل ترتیب قرارگیری پارامترها درون پشته یا ثبات‌ها می‌شود و این‌که بعد از به اتمام رسیدن کار تابع آیا فراخوانی کننده تابع یا تابع فراخوانی شده مسئول پاک‌سازی پشته است یا خیر.

قراردادهای فراخوانی توابع مطابق با نوع کامپایلر زبان‌های برنامه‌نویسی متفاوت هستند و اغلب تفاوت‌های ظریفی در چگونگی پیاده‌سازی این قراردادها توسط کامپایلرها وجود دارد. به هر صورت، هنگام استفاده از رابط‌های برنامه‌نویسی (API) که به منظور سازگاری بین نسخه‌های مختلف ویندوز بطور یک نواخت پیاده‌سازی می‌شوند، شما باید یک سری قواعد خاص را دنبال کنید. (در قسمت هفتم درباره این موضوع بحث خواهیم کرد). شایان ذکر است، در این قسمت از شبه کد لیست ۱۶ برای تشریح فراخوانی هر قرارداد استفاده خواهیم کرد.

```
1   // Code snippet by Milad Kahsari Alhadi
2
3   #include <windows.h>
4   #include <iostream>
5
6   int test(int x, int y, int z)
7   {
8       return x + y + z;
9   }
10
11  int main(int argc, const char* argv[])
12  {
13      int a = 1 , b = 2, c = 3, ret = 0;
14
15      ret = test(a, b, c);
16
17      return 0;
18  }
```

لیست ۱۶: شبه کد برای فراخوانی یک تابع

سه تا از رایج‌ترین قراردادهای فراخوانی که با آن‌ها مواجه می‌شوید cdecl ، stdcall و fastcall هستند. درباره تفاوت‌های کلیدی میان آن‌ها در بخش زیر بحث خواهیم کرد.

نکته: گرچه قراردادهای مختلف می‌توانند به شیوه مختلفی توسط کامپایلرها پیاده‌سازی شوند. با این حال ما روی رایج‌ترین راه استفاده از آن‌ها متمرکز خواهیم شد، با ما همراه باشید.

قرارداد Cdecl به منظور فراخوانی پارامترهای توابع

cdecl یکی از مشهورترین قراردادهایی است که ما در قسمت چهارم هنگام معرفی پشته و فراخوانی توابع آن را تشریح کردیم. در قرارداد فراخوانی توابع cdecl، پارامترهای عبور داده شده به توابع از راست به چپ درون پشته قرار می‌گیرند و پس از اتمام وظیفه اجرایی تابع، فراخواننده تابع پشته را پاک‌سازی کرده و مقدار بازگشتی تابع مذکور را به درون ثبات EAX منتقل می‌کند. لیست ۱۷، دستورالعمل‌های دیزاسمبلی کد لیست ۱۶ را که با استفاده از قرارداد CDECL کامپایل شده بودند، نمایش می‌دهد.

```
; int __cdecl main(int argc, const char **argv, const char **envp)
_main proc near

var_10= dword ptr -10h
var_C= dword ptr -0Ch
var_8= dword ptr -8
var_4= dword ptr -4
argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h

push    ebp
mov     ebp, esp
sub     esp, 10h
mov     [ebp+var_C], 1
mov     [ebp+var_8], 2
mov     [ebp+var_4], 3
mov     [ebp+var_10], 0
mov     eax, [ebp+var_4]
push    eax
mov     ecx, [ebp+var_8]
push    ecx
mov     edx, [ebp+var_C]
push    edx
call    sub_401000
add     esp, 0Ch
mov     [ebp+var_10], eax
xor     eax, eax
mov     esp, ebp
pop     ebp
retn
_main endp
```

لیست ۱۷: فراخوانی تابع **cdecl**

همانطور که مشاهده می‌کنید با استفاده از دستورالعمل add esp, 0Ch پشته توسط فراخواننده[1] تابع test که اینجا همان تابع main است، پشته پاکسازی شده است. همچنین در این مثال، با توجه به دستورالعمل push [ebp+var_4] متوجه می‌شویم که پارامترها از راست به چپ به درون پشته قرار داده شده‌اند.

[1] Caller

قرارداد Stdcall به منظور فراخوانی پارامترهای توابع

شهرت قرارداد فراخوانی توابع stdcall مشابه شهرت cdecl می‌باشد، جزء این‌که stdcall برای پاک‌سازی پشته پس از به اتمام رسیدن وظیفه عملیاتی تابع، خود تابع فراخوانی شده[1] پشته را پاکسازی می‌کند. از این رو، اگر قرارداد stdcall استفاده شود، به دستورالعمل add که در لیست ۱۷ به آن اشاره شد، نیاز نخواهد بود. زیرا تابع فراخوانی شده خود پاسخگو پاک‌سازی تابع است. تابع test در لیست ۱۶ تحت قرارداد stdcall به شکل متفاوتی کامپایل خواهد شد، بدین دلیل که این تابع باید خود پشته را پاک‌سازی کند.

قرارداد Stdcall، قرارداد استاندارد فراخوانی رابطهای برنامه‌نویسی ویندوز (Windows API) است. شایان ذکر است، هر کدی که رابطهای برنامه‌نویسی ویندوز را فراخوانی کند، از آنجایکه مسئولیت کتابخانه‌های پیوندی پویا (DLLs) است که کد توابع رابطهای برنامه‌نویسی ویندوز را پیاده‌سازی کنند، لذا به پاک‌سازی پشته نیاز نخواهد داشت.

قرارداد Fastcall به منظور فراخوانی پارامترهای توابع

قرارداد فراخوانی Fastcall در کامپیالرها متفاوت است، اما در حالت معمول بطور مشابهی در تمامی حالات کار می‌کند. در Fastcall، چند پارامتر اول (معمولا دو تا پارامتر اول) به ثبات‌ها عبور داده می‌شوند. ثبات‌هایی که عموما بدین منظور استفاده می‌شوند ثبات‌های Eax و Ecx هستند. سپس پارامترهای اضافی از راست به چپ بارگذاری شده و اگر نیاز به پاک‌سازی پشته باشد، فراخوانی‌کننده تابع معمولا پاسخگوی پاک‌سازی پشته است. اغلب اوقات استفاده از این قرارداد نسب به بقیه کار آمدتر است، زیرا کد نیاز ندارد زیاد درگیر پشته شود.

دستورالعمل Push در مقابل Mov

علاوه بر استفاده از قراردادهای مختلف که تا به الان تشریح شد، کامپایلرها ممکن است از دستورالعمل‌های مختلفی برای انجام عملیات‌های مشابه استفادهه کنند: به عنوان مثال، گاهی اوقات کامپایلر تصمیم می‌گیرد از دستورالعمل mov به جای push برای قراردادن آیتم‌ها درون پشته استفاده کند. لیست ۱۸، یک کد C

[1] Callee

برای فراخوانی یک تابع را نمایش می‌دهد. تابع adder دو پارامتر را با هم جمع کرده و نتیجه را بازگشت می‌دهد. تابع اصلی تابع adder را فراخوانی می‌کند و نتیجه را در خروجی چاپ می‌کند.

```
1  // Code snippet by Milad Kahsari Alhadi
2
3  #include <windows.h>
4  #include <iostream>
5
6  int adder(int a, int b)
7  {
8      return a + b;
9  }
10
11 int main(int argc, const char* argv[])
12 {
13     int x = 1;
14     int y = 2;
15
16     printf("the function returned the number %d\n", adder(x, y));
17
18     return 0;
19 }
```

لیست ۱۸: کد C برای فراخوانی یک تابع

کد دیزاسمبلی برای تابع adder در سراسر کامپایلرها ثابت است که در لیست ۱۹ به نمایش گذاشته شده است. همان‌طورکه مشاهده می‌کنید، این کد دو پارامتر، arg_0 با arg_4 را جمع کرده و نتیجه را در ثبات EAX ذخیره می‌سازد. همان‌طور که در قسمت چهارم این سلسله مقالات بحث شده از EAX برای ذخیره‌سازی مقادیر بازگشتی استفاده می‌شود.

```
00401730        push    ebp
00401731        mov     ebp, esp
00401733        mov     eax, [ebp+arg_0]
00401736        add     eax, [ebp+arg_4]
00401739        pop     ebp
0040173A        retn
```

لیست ۱۹: کد اسمبلی برای تابع adder در لیست ۱۸

جدول ۱ قراردادهای مختلف فراخوانی که توسط دو کامپایلر مایکروسافت و کامپایلر GNU استفاده شده است را نمایش می‌دهد. در سمت چپ، پارامترها برای توابع adder و printf قبل از فراخوانی در پشته قرار داده شده‌اند و در سمت راست، قبل از فراخوانی تابع پارامترها به درون پشته منتقل شده‌اند. شما باید به عنوان تحلیلگران بدافزار آمادگی مواجه شدن با هر دو نوع خروجی این کامپایلرها را داشته باشید، زیرا تحلیلگران

بدافزار کنترلی روی کامپایلرها نخواهند داشت. به عنوان مثال، یک دستورالعمل در سمت راست مطابق با هیچ کدوم از دستورالعمل‌های موجود در سمت چپ نیست. مثلا دستورالعملی که اشاره‌گر را بازیابی می‌کند، در سمت راست نیاز نیست زیرا اشاره‌گر پشته هیچ وقت تغییر نمی‌کند.

نکته: این نکته یادتان باشد، حتی زمانی‌که کامپایلر مشابهی استفاده می‌شود، مطابق با تنظیمات و گزینه‌های کامپایلر می‌تواند در فراخوانی قراردادهای توابع تفاوت وجود داشته باشد.

جدول ۱: کد اسمبلی فراخوانی تابع با دو قرارداد فراخوانی متفاوت

Visual Studio version			GCC version		
00401746	mov	[ebp+var_4], 1	00401085	mov	[ebp+var_4], 1
0040174D	mov	[ebp+var_8], 2	0040108C	mov	[ebp+var_8], 2
00401754	mov	eax, [ebp+var_8]	00401093	mov	eax, [ebp+var_8]
00401757	push	eax	00401096	mov	[esp+4], eax
00401758	mov	ecx, [ebp+var_4]	0040109A	mov	eax, [ebp+var_4]
0040175B	push	ecx	0040109D	mov	[esp], eax
0040175C	call	adder	004010A0	call	adder
00401761	**add**	**esp, 8**			
00401764	push	eax	004010A5	mov	[esp+4], eax
00401765	push	offset TheFunctionRet	004010A9	mov	[esp], offset TheFunctionRet
0040176A	call	ds:printf	004010B0	call	printf

تحلیل عبارات Switch

عبارات switch توسط برنامه‌نویسان (نویسندگان بدافزار) برای تصمیم‌گیری بر مبنای یک کاراکتر یا یک عدد صحیح مورد استفاده قرار می‌گیرد. به عنوان مثال، مجموعه عملیات‌های درهای‌پشتی معمولا با استفاده از یک بایت انتخاب می‌شوند. با این حال، عبارت‌های Switch با دو روش عمومی یعنی استفاده از سبک IF یا استفاده از جداول پرش کامپایل می‌شوند. در ادامه این دو روش را مورد بررسی قرار خواهیم داد.

سبک if

لیست ۲۰ یک مثال ساده از عبارت switch نمایش می‌دهد که از متغیر i به عنوان عملوند شرط استفاده می‌کند و مطابق با مقدار متغیر i، کد درون یکی از حالت‌ها (Case) اجرا می‌گردد.

```c
// Code snippet by Milad Kahsari Alhadi

#include <windows.h>
#include <iostream>

int main(int argc, const char* argv[])
{
    int i = 1;

    switch (i)
    {
    case 1:
        printf("i = %d", i + 1);
        break;
    case 2:
        printf("i = %d", i + 2);
        break;
    case 3:
        printf("i = %d", i + 3);
        break;
    default:
        break;
    }
    return 0;
}
```

لیست ۲۰: کد C برای یک عبارت switch سه گزینه‌ای

کد اسمبلی حاصل از کامپایل عبارت switch در لیست ۲۰ را می‌توانید در لیست ۲۱ مشاهده کنید. لیست ۲۱ شامل یک مجموعه از پرش‌های شرطی است که میان شماره‌ها ۱ و ۲ مشخص شده‌اند. پرش‌های شرطی توسط مقایسه‌هایی که مستقیما قبل از هر پرش صورت می‌گیرند، رخ می‌دهند. عبارت switch سه گزینه دارد که با شماره‌های ۳، ۴ و ۵ نمایش داده شده‌اند. این قسمت‌های کد از مابقی مجزا هستند زیرا پرش‌های غیر شرطی به پایان لیست به شما می‌روند. (احتمالا فهمیده‌اید که عبارات سوئیچ با استفاده از گراف نمایش داده شده در تصویر ۳ بسیار راحت قابل فهم هستند).

```
00401013         cmp     [ebp+var_8], 1
00401017         jz      short loc_401027 ❶
00401019         cmp     [ebp+var_8], 2
0040101D         jz      short loc_40103D
0040101F         cmp     [ebp+var_8], 3
00401023         jz      short loc_401053
00401025         jmp     short loc_401067 ❷
00401027 loc_401027:
00401027         mov     ecx, [ebp+var_4] ❸
0040102A         add     ecx, 1
0040102D         push    ecx
0040102E         push    offset unk_40C000 ; i = %d
00401033         call    printf
00401038         add     esp, 8
0040103B         jmp     short loc_401067
0040103D loc_40103D:
0040103D         mov     edx, [ebp+var_4] ❹
00401040         add     edx, 2
00401043         push    edx
00401044         push    offset unk_40C004 ; i = %d
00401049         call    printf
0040104E         add     esp, 8
00401051         jmp     short loc_401067
00401053 loc_401053:
00401053         mov     eax, [ebp+var_4] ❺
00401056         add     eax, 3
00401059         push    eax
0040105A         push    offset unk_40C008 ; i = %d
0040105F         call    printf
00401064         add     esp, 8
```

لیست ۲۱: کد اسمبلی برای عبارت سوئیچ مثال لیست ۲۰

تصویر ۳ هر یک از گزینه‌های switch را با تقسیم کدهای اجرا شده به سمت پایین مجزاسازی کرده است. سه تا از جعبه‌های نمایش داده شده در عکس با شماره‌های ۱، ۲ و ۳ برچسب‌گذاری شده‌اند که مطابق با سه حالت case دستور switch هستند. توجه کنید که تمامی جعبه‌ها به جعبه پایینی خاتمه پیدا می‌کنند.

با این حال، باید قادر به استفاده از این گراف برای مشاهده سه کد بررسی شده هنگام بزرگتر بودن var_8 از ۳ باشید. همچنین قابل ذکر است، درک این‌که کدهای تولید شده در فرآیند دیزاسمبلی مربوط به یک عبارات switch یا یک سلسله مراتب از دستورات if هستند، بسیار دشوار است. زیرا کدهای کامپایل شده عبارت‌های switch و عبارت‌های if مشابه به نظر می‌رسند و هر دوی آن‌ها شامل مجموعه‌ای از دستورالعمل‌های cmp و jcc می‌شوند. هنگامی انجام فرآیند دیزاسمبلی، ممکن است همیشه نتوانید به کد

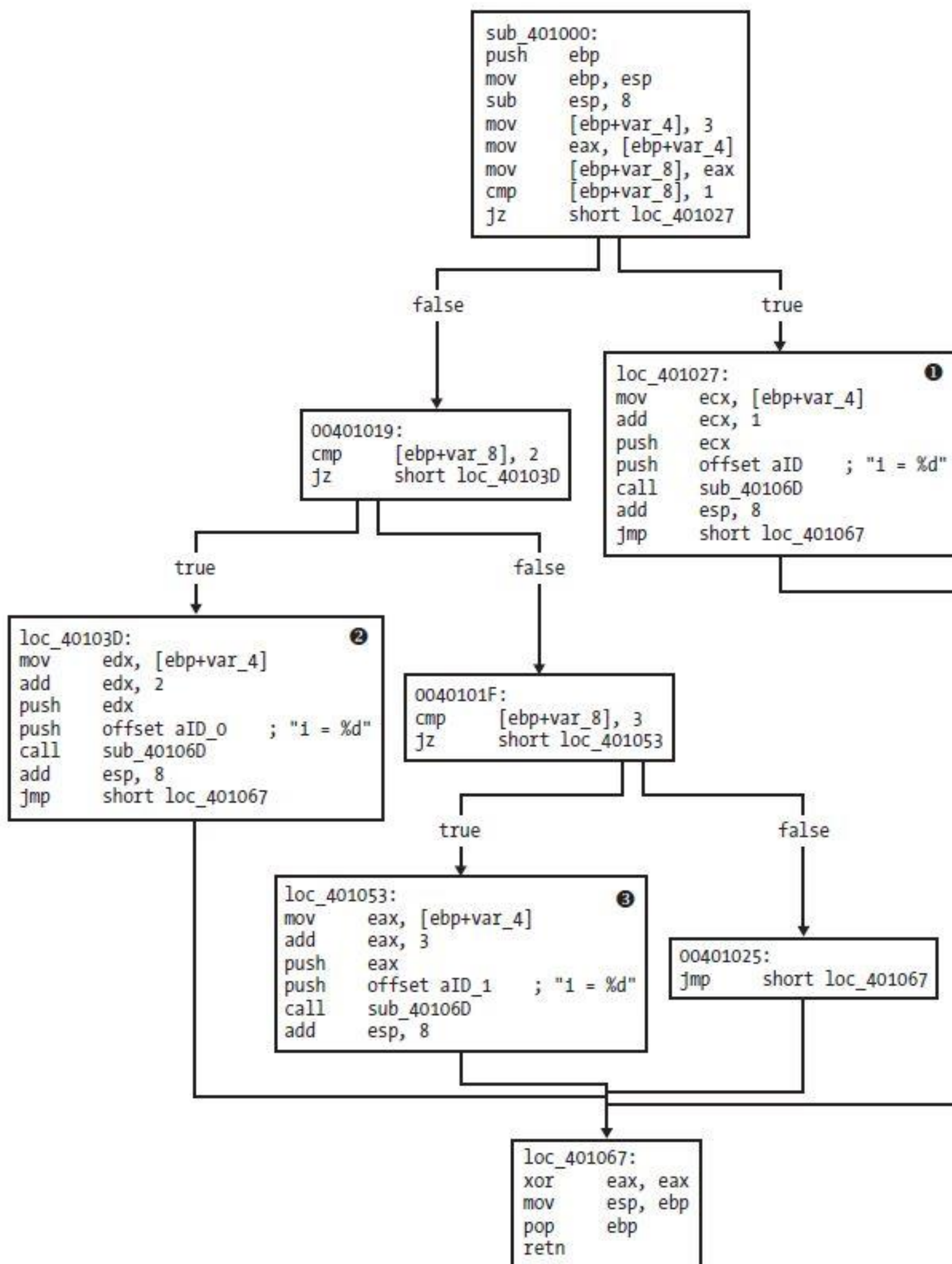
اصلی بازگردید، زیراکه راه‌های گوناگونی برای نمایش عملیات‌های مشابه در زبان اسمبلی وجود دارد که تمامی آن‌ها قانونی و مشابه یکدیگر هستند.

جدول پرش

مثال دیزاسمبلی بعدی به صورت رایج در عبارت‌های switch ادامه دار و گسترده پیدا می‌شود. کامپایلر کد را بهینه‌سازی می‌کند زیرا به دستورالعمل‌های مقایسه‌ای بسیاری نیاز نداشته باشد. به عنوان مثال، اگر در لیست ۲۰ مقدار i برابر با ۳ بود، سه مقایسه قبل از case سوم صورت نمی‌گیرند. در لیست ۲۲، ما یک case دیگر به لیست ۲۰ اضافه کردیم (همان‌طور که با مقایسه دو لیست می‌توانید تفاوت‌ها را دریابید) اما کد اسمبلی کاملا متفاوت است.

```
6   int main(int argc, const char* argv[])
7   {
8       int i = 1;
9   
10      switch (i)
11      {
12      case 1:
13          printf("i = %d", i + 1);
14          break;
15      case 2:
16          printf("i = %d", i + 2);
17          break;
18      case 3:
19          printf("i = %d", i + 3);
20          break;
21      case 4:
22          printf("i = %d", i + 3);
23          break;
24      default:
25          break;
26      }
27  
28      return 0;
29  }
```

لیست ۲۲: کد C برای یک عبارت switch چهار گزینه‌ای



تصویر ۳: گراف دیزاسمبلی عبارت switch سبک if مثال لیست ۲۱

homeland security
kaipod
کیان امنیت

کد اسمبلی در لیست ۲۳ از یک جدول پرش استفاده می‌کند (شماره ۲) که آفست‌ها را به علاوه محل‌های حافظه تعریف می‌کند. شایان ذکر است، متغیر عبارت switch به عنوان یک ایندکس در جدول پرش استفاده شده است.

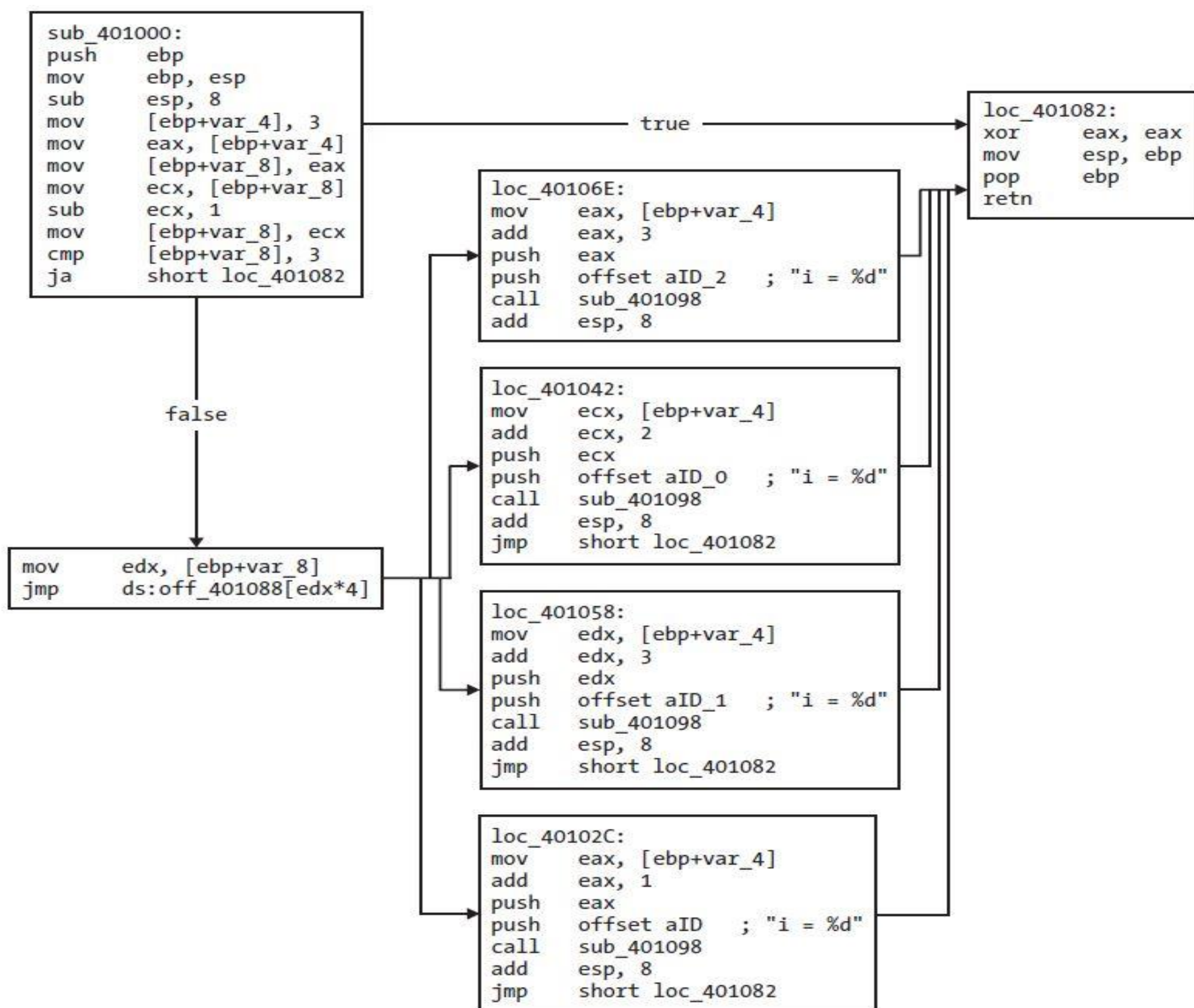
در این مثال، ثبات ecx شامل متغیر switch می‌شود و از آن ۱ واحد در خط اول کم می‌شود. در کد C محدوده switch از ۱ تا ۴ است اما کد اسمبلی باید این محدوده را از ۰ تا ۳ تنظیم کند که جدول پرش بتواند بطور مناسب ایندکس شود. دستورالعمل پرش (شماره ۱) جایی است که هدف مبتنی بر جدول پرش در آنجا قرار دارد.

در این دستورالعمل پرش، ثبات edx با مقدار ۴ ضرب شده و با مقدار پایه جدول پرش (0x401088) جمع می‌شود تا مشخص شود به کدام case باید پرش صورت گیرد. شایان ذکر است، این دستورالعمل با ۴ ضرب می‌شود زیرا هر موجودیت در جدول پرش یک آدرس است که ۴ بایت اندازه دارد.

```
00401016         sub     ecx, 1
00401019         mov     [ebp+var_8], ecx
0040101C         cmp     [ebp+var_8], 3
00401020         ja      short loc_401082
00401022         mov     edx, [ebp+var_8]
00401025         jmp     ds:off_401088[edx*4] ❶
0040102C    loc_40102C:
               ...
00401040         jmp     short loc_401082
00401042    loc_401042:
               ...
00401056         jmp     short loc_401082
00401058    loc_401058:
               ...
0040106C         jmp     short loc_401082
0040106E    loc_40106E:
               ...
00401082    loc_401082:
00401082         xor     eax, eax
00401084         mov     esp, ebp
00401086         pop     ebp
00401087         retn
00401087    _main   endp
00401088   ❷off_401088   dd offset loc_40102C
0040108C                 dd offset loc_401042
00401090                 dd offset loc_401058
00401094                 dd offset loc_40106E
```

لیست ۲۳: کد اسمبلی برای عبارت switch مثال لیست ۲۲

گراف نمایش داده شده در تصویر ۴ برای این نوع عبارت switch خیلی واضح تر از خروجی نمایش داده شده در پنجره دیزاسمبلی استاندارد است.



تصویر ٤: گراف دیزاسمبلی مثال جدول پرش عبارت switch

همان‌طور که مشاهده می‌کنید، هر یک از چهار case بطور خیلی واضح به قسمت‌های جدا شده شکسته شده‌اند. هر یک از این بخش‌ها قسمتی از کد را نمایش می‌دهند که پس از تصمیم‌گیری باید به آن‌ها پرش شود. توجه کنید همه این جعبه‌ها و همچنین جعبه مقداردهی اولیه به جعبه سمت راست پایان داده می‌شوند.

homeland security kaipod
کیان امنیت کایپد

دیزاسمبلی آرایه‌ها

آرایه‌ها توسط برنامه‌نویسان برای تعریف مجموعه سلسله مراتبی از داده‌های مشابه مورد استفاده قرار می‌گیرند. بدافزارها گاهی اوقات از یک آرایه اشاره‌گر به رشته‌ها که شامل چندین نام میزبان می‌شود به عنوان گزینه‌هایی برای برقراری ارتباط استفاده می‌کنند. لیست ۲۴ دو آرایه استفاده شده توسط یک برنامه را نمایش می‌دهد، هر دو آرایه در طی تکرار از طریق حلقه for تنظیم می‌شوند. آرایه a به صورت محلی تعریف شده است و آرایه b به صورت عمومی تعریف شده است. این تعاریف در کد اسمبلی تولید شده توسط کامپایلر تاثیر گذار است.

```
1   // Code snippet by Milad Kahsari Alhadi
2
3   #include <windows.h>
4   #include <iostream>
5
6   int b[5] = { 123,87,487,7,978 };
7
8   int main(int argc, const char* argv[])
9   {
10
11      int i;
12      int a[5];
13
14      for (i = 0; i < 5; i++)
15      {
16          a[i] = i;
17          b[i] = i;
18      }
19
20      return 0;
21  }
```

لیست ۲٤: کد C برای یک آرایه

در اسمبلی، آرایه‌ها توسط یک آدرس پایه به عنوان نقطه شروع در دسترس قرار می‌گیرند. سایز هر عنصر همیشه آشکار است که می‌توانید با مشاهده چگونگی ایندکس آرایه اندازه آن را مشخص سازید. لیست ۲۵ یک کد اسمبلی برای لیست ۲۴ را نمایش می‌دهد.

```
00401006         mov     [ebp+var_18], 0
0040100D         jmp     short loc_401018
0040100F loc_40100F:
0040100F         mov     eax, [ebp+var_18]
00401012         add     eax, 1
00401015         mov     [ebp+var_18], eax
00401018 loc_401018:
00401018         cmp     [ebp+var_18], 5
0040101C         jge     short loc_401037
0040101E         mov     ecx, [ebp+var_18]
00401021         mov     edx, [ebp+var_18]
00401024         mov     [ebp+ecx*4+var_14], edx ❶
00401028         mov     eax, [ebp+var_18]
0040102B         mov     ecx, [ebp+var_18]
0040102E         mov     dword_40A000[ecx*4], eax ❷
00401035         jmp     short loc_40100F
```

لیست ۲۵: کد اسمبلی برای آرایه لیست ۲٤

در این لیست، آدرس پایه برای آرایه b مطابق با dword_40A000 است و آدرس پایه برای آرایه a مطابق با var_14 است. از آنجایی که هر دو این آرایه ها از نوع عدد صحیح هستند، هر عنصر ۴ بایت اندازه دارد، گرچه (شماره ۱) و (شماره ۲) دستورالعملی متفاوت برای دسترسی به دو آرایه متفاوت هستند. در هر دو حالت از ثبات ecx به عنوان ایندکس استفاده می‌شود که در ادامه با مقدار عددی ۴ ضرب می‌شود. همانطور که قبلا ذکر کردیم، مقدار عددی ۴ اندازه هر بایت از عنصرهای آرایه است و در پایان مقدار نتیجه با آدرس پایه آرایه برای دسترسی به عنصرهای مناسب آرایه جمع می‌شود.

شناسایی استراکچرها[1]

استراکچرها یا Structures مشابه آرایه‌ها هستند، اما از انواع داده مختلفی تشکیل می‌شوند. استراکچرها عموما توسط نویسندگان بدافزار برای گروه‌بندی اطلاعات مورد استفاده قرار می‌گیرند. گاهی اوقات استفاده از استراکچرها به جای نگهداری متغیرهای مختلف به صورت جداگانه ساده‌تر است، مخصوصا اگر نیاز باشد چندین تابع به یک گروه مشابه از متغیرها دسترسی پیدا کنند (توابع API ویندوز اغلب از استراکچرها استفاده می‌کنند که باید توسط برنامه فراخوانی کننده ایجاد و نگهداری شوند).

در لیست ۲۶، ما یک استراکچر (شماره ۱) تعریف کردیم که یک آرایه از اعداد صحیح، یک کاراکتر و یک متغیر از نوع double ایجاد می‌کند. سپس در تابع Main برای این ساختمان از Heap حافظه گرفته‌ایم

[1] Identifying Structs

و در ادامه ساختمان را به تابع test عبور داده‌ایم. شایان ذکر است، ساختمان gms (شماره ۲) به عنوان یک متغیر عمومی تعریف شده است.

```
struct my_structure { ❶
     int x[5];
     char y;
     double z;
};

struct my_structure *gms; ❷

void test(struct my_structure *q)
{
     int i;
     q->y = 'a';
     q->z = 15.6;
     for(i = 0; i<5; i++){
           q->x[i] = i;
     }
}

void main()
{
     gms = (struct my_structure *) malloc(
     sizeof(struct my_structure));
     test(gms);
}
```

لیست ۲۶: کد C برای مثال یک استراکچر

استراکچرها (مانند آرایه‌ها) توسط یک آدرس پایه به عنوان نقطه شروع در دسترس قرار می‌گیرند. قابل ذکر است توانایی شما در شناسایی استراکچرها می‌تواند یک تاثیر فوق العاده در تجزیه و تحلیل بدافزار بگذارد.

لیست ۲۷ تابع اصلی کد موجود در لیست ۲۶ را نمایش می‌دهد که دیزاسمبل شده است. از آنجایی‌که struct gms یک متغیر گلوبال است، آدرس پایه آن محل حافظه dword_40EA30 خواهد بود که در لیست ۲۷ نمایش داده شده است. آدرس پایه این استراکچر به تابع sub_401000(test) از طریق push eax (شماره ۱) عبور داده شده است.

```
00401050         push    ebp
00401051         mov     ebp, esp
00401053         push    20h
00401055         call    malloc
0040105A         add     esp, 4
0040105D         mov     dword_40EA30, eax
00401062         mov     eax, dword_40EA30
00401067         push    eax ❶
00401068         call    sub_401000
0040106D         add     esp, 4
00401070         xor     eax, eax
00401072         pop     ebp
00401073         retn
```

لیست ۲۷: کد اسمبلی برای تابع اصلی در استراکچر مثال لیست ۲۶

لیست ۲۸ کد دیزاسمبل شده تابع test درون لیست ۲۶ را نمایش می‌دهد. پارامتر arg_0 آدرس پایه استراکچر است. آفست 0x14 کاراکتر درون ساختمان را ذخیره می‌کند و 0x61 مطابق با حرف a در اسکی است.

```
00401000         push    ebp
00401001         mov     ebp, esp
00401003         push    ecx
00401004         mov     eax,[ebp+arg_0]
00401007         mov     byte ptr [eax+14h], 61h
0040100B         mov     ecx, [ebp+arg_0]
0040100E         fld     ds:dbl_40B120 ❶
00401014         fstp    qword ptr [ecx+18h]
00401017         mov     [ebp+var_4], 0
0040101E         jmp     short loc_401029
00401020 loc_401020:
00401020         mov     edx,[ebp+var_4]
00401023         add     edx, 1
00401026         mov     [ebp+var_4], edx
00401029 loc_401029:
00401029         cmp     [ebp+var_4], 5
0040102D         jge     short loc_40103D
0040102F         mov     eax,[ebp+var_4]
00401032         mov     ecx,[ebp+arg_0]
00401035         mov     edx,[ebp+var_4]
00401038         mov     [ecx+eax*4],edx ❷
0040103B         jmp     short loc_401020
0040103D loc_40103D:
0040103D         mov     esp, ebp
0040103F         pop     ebp
00401040         retn
```

لیست ۲۸: کد اسمبلی برای تابع test موجود در استراکچر مثال لیست ۲۶

می‌توانیم بگوییم آفست 0x18 یک double است، زیرا به عنوان قسمتی از یک دستورالعمل اعشاری استفاده شده است (شماره ۱). همچنین می‌توانیم با بررسی حلقه for و جایی‌که این آفست‌ها در دسترس قرار می‌گیرند بگوییم که اعداد صحیح به درون آفست 4، 0، 8، 0xC و 0x10 منتقل شده‌اند. همچنین در این تجزیه و تحلیل می‌توانیم به محتویات این استراکچرها پی ببریم. شایان ذکر است، در دیزاسمبلر IDA Pro می‌توانید استراکچر ایجاد کنید و به آن‌ها با کلید T حافظه تخصیص دهید. انجام این عملیات دستورالعمل mov [eax+14h], 61h را به mov [eax + my_structure.y], 61h تغییر می‌دهد.

حروف راحت قابل خواندن هستند و علامت‌گذاری ساختمان‌ها اغلب می‌تواند به شما در درک کدهای دیزاسمبلی کمک شایانی کند، مخصوصا اگر بطور مداوم استراکچرهای استفاده شده را مشاهده کنید. برای استفاده از کلید T به صورت موثر در این مثال، نیاز دارید استراکچر my_structure را با استفاده از Structures Window در دیزاسمبلر IDA Pro ایجاد کنید. این فرایند می‌تواند بسیار خسته کننده باشد، اما این موضوع همچنین می‌تواند برای استراکچرهایی که پیوسته با آن‌ها مواجه می‌شوید، بسیار مفید باشد.

تجزیه و تحلیل لیست‌های پیوندی پیمایشی[1]

یک لیست پیوندی یک ساختمان داده است که شامل یک مجموعه از داده‌های ذخیره شده می‌شود و ترتیب خطی عناصر داده‌ای در آن توسط اشاره‌گرها تعیین می‌شود. مزئیت اصلی استفاده از لیست‌های پیوندی به جای آرایه‌ها این است که ترتیب عنصرهای پیوندی از ترتیب عناصر داده ذخیره شده در دیسک یا حافظه می‌تواند متفاوت باشد. از این رو لیست‌های پیوندی به شما اجازه می‌دهند در هر نقطه که می خواهید یک گره اضافه یا آن را حذف کنید.

لیست ۲۹ یک مثال با زبان C از لیست‌های پیوندی و پیمایش آن را نشان می‌دهد. این لیست پیوندی شامل یک مجموعه از گره‌ها می‌شود که با pnode نام‌گذاری شده‌اند و با دو حلقه دستکاری می‌شوند. اولین حلقه (شماره ۱) ده گره ایجاد کرده و آن‌ها را با اطلاعات پر می‌کند. حلقه دوم (شماره ۲) همه داده‌های ثبت شده را تکرار می‌کند و محتویات آن‌ها را در خروجی چاپ می‌کند.

[1] Analyzing Linked List Traversal

```
struct node
{
   int x;
   struct node * next;
};

typedef struct node pnode;

void main()
{
   pnode * curr, * head;
   int i;
   head = NULL;

   for(i=1;i<=10;i++) ❶
   {
      curr = (pnode *)malloc(sizeof(pnode));
      curr->x = i;
      curr->next  = head;
      head = curr;
   }

   curr = head;

   while(curr) ❷
   {
      printf("%d\n", curr->x);
      curr = curr->next ;
   }
}
```

لیست ۲۹: کد C برای یک لیست پیوندی پیمایشی

بهترین راه برای فهمیدن کد دیزاسمبل شده برنامه لیست ۲۹ شناسایی دو ساختمان درون تابع main مخصوصا ساختمان curr است. توانایی شما در شناسایی کدها این ساختمان باعث می‌شود کار تحلیلتان آسان‌تر شود.

در لیست ۳۰، ما در گام اول حلقه for را شناسایی کرده‌ایم. در لیست ۳۰ متغیر var_c دقیقا برابر متغیر i و متغیر var_8 برابر متغیر head و var_4 برابر با متغیر curr در لیست ۲۹ است. شایان ذکر است، متغیر var_4 یک اشاره‌گر به یک ساختمان با دو متغیر می‌باشد که به آن مقادیر (شماره ۱ و شماره ۲) تخصیص داده شده است. حلقه تکرار while (شماره ۳ تا شماره ۵) تکرارکننده لیست پیوندی را اجرا می‌کند و درون حلقه، متغیر var_4 (شماره ۴) با رکورد بعدی در لیست تنظیم خواهد شد.

```
0040106A         mov     [ebp+var_8], 0
00401071         mov     [ebp+var_C], 1
00401078
00401078 loc_401078:
00401078         cmp     [ebp+var_C], 0Ah
0040107C         jg      short loc_4010AB
0040107E         mov     [esp+18h+var_18], 8
00401085         call    malloc
0040108A         mov     [ebp+var_4], eax
0040108D         mov     edx, [ebp+var_4]
00401090         mov     eax, [ebp+var_C]
00401093         mov     [edx], eax ❶
00401095         mov     edx, [ebp+var_4]
00401098         mov     eax, [ebp+var_8]
0040109B         mov     [edx+4], eax ❷
0040109E         mov     eax, [ebp+var_4]
004010A1         mov     [ebp+var_8], eax
004010A4         lea     eax, [ebp+var_C]
004010A7         inc     dword ptr [eax]
004010A9         jmp     short loc_401078
004010AB loc_4010AB:
004010AB         mov     eax, [ebp+var_8]
004010AE         mov     [ebp+var_4], eax
004010B1
004010B1 loc_4010B1:
004010B1         cmp     [ebp+var_4], 0 ❸
004010B5         jz      short locret_4010D7
004010B7         mov     eax, [ebp+var_4]
004010BA         mov     eax, [eax]
004010BC         mov     [esp+18h+var_14], eax
004010C0         mov     [esp+18h+var_18], offset aD ; "%d\n"
004010C7         call    printf
004010CC         mov     eax, [ebp+var_4]
004010CF         mov     eax, [eax+4]
004010D2         mov     [ebp+var_4], eax ❹
004010D5         jmp     short loc_4010B1 ❺
```

لیست ۳۰: کد اسمبلی برای لیست پیوندی پیمایشی مثال موجود در لیست ۲۹

برای شناسایی یک لیست پیوندی، ابتدا باید برخی از اشیاء که حاوی اشاره‌گر هستند و به یک شی دیگر از همان نوع اشاره می‌کنند را شناسایی کنید. طبیعت بازگشتی اشیاء چیزی است که لیست های پیوندی را مشخص می‌کند و این چیزی است که شما برای شناسایی لیست‌های پیوندی در دیزاسمبلی به آن نیاز دارید.

homeland security kaipod
کیان امنیت کایپد

نتیجه‌گیری

این قسمت برای نمایش یک عملیات ثابت در تحلیل بدافزار به منظور جدا ساختن تحلیلگر از جزییات، طراحی شده بود. با این حال، همواره یادتان باشد، خود را گرفتار جزییات سطح پایین برنامه نکنید، اما توانایی خود را در شناسایی آنچه که کدها در سطح بالا انجام می‌دهند را توسعه دهید.

ما تمامی ساختمان‌های مهم کدهای زبان برنامه‌نویسی C را در اسمبلی به شما نمایش دادیم تا به شما در شناسایی بیشتر ساختمان‌های رایج در طی تجزیه و تحلیل کمک کرده باشیم. همچنین ما دو مثال که در آن‌ها حالت‌های مختلف تصمیم‌گیری کامپایلر برای کامپایل ساختمان‌ها و فراخوانی توابع (هنگامی‌که از یک کامپایلر کاملا متفاوت استفاده می‌شود) را نشان دادیم. توسعه دانش خود در این زمینه‌ها به شما کمک می‌کند؛ هرگاه با ساختمان‌های جدیدی رو به رو شدید آن‌ها را شناسایی کنید.