توضیحاتی پیرامون کار
با برنامه Process Stalker

در مبحث آنالیز داده ها و ردیابیِ کد یا Code Tracing (یکی از مباحث Disassembly) به ناگاه در فضای اینترنت به عبارت **Process Stalking** برخوردم. عملیات یا ابزار Process Stalking یکی از اهدافی است که به صورت یک update یا plug-in برای IDA Pro در نظر گرفته شده است. این ابزار طبق شواهد و تحقیقات انجام شده، توسط گروه **iDEFENSE** رشد و گسترش داده شده و بنیان گذار این روشِ تجزیه و تحلیل داده ها، آقای پدرام امینی می باشد، یکی از افرادی که در بروز رسانی و پیدایش اولیه **IDA Pro** نیز سهم بسزایی داشته اند. همچنین باید گوشزد کرد که تعداد کمی از اسکریپت هایی که توسط گروه **iDEFENSE** ارائه شده اند، دارای مشکلاتی بوده که با تلاش، از آنها رفع مشکل شد. در صورت نیاز آماده به ارائه اسکریپت های اشکال زدا شده نیز هستم. نیز باید خاطر نشان ساخت که این مقاله با اجازه قبلی از آقای امینی ترجمه شده است.

مقدمه

Process Stalking کلمه ای است که برای توصیف فرآیند ترکیب شده ای از نمایه زمان-اجرا[1]، نگاشت و ردیابی حالت[2] به کار می رود. با شامل بودن رشته ای از ابزار و اسکریپت ها، هدف یک stalkِ موفق، فراهم کردن یک رابط لذت بخش برای مهندس معکوس است تا بتواند با آن رابط به داده های فیلترشده، معنی دار و ردیابی سطح-بلوک در زمان-اجرا[3] برسد.

رشته Process Stalking به سه بخش اصلی تقسیم می شود: یک پلاگینِ IDA Pro، یک ابزار ردیابی[4] مستقل و یک سری از اسکریپت های پیتون برای ایجاد[5] فایل های گرافی GML حدواسط. تعاریف گراف های تولید شده GML، به منظور استفاده از آنها با یک ابزار بصریِ گرافیکیِ کاملا قابل دسترس طراحی شده اند.

پلاگین IDA-Pro

برای نصب پلاگین IDA، نسخه مناسبی از process_stalker.plw را در شاخه plugins واقع در شاخه ای که IDA Pro در آن نصب شده است، کپی کنید. در صورتی که عملیات نصب به درستی انجام شده باشد، هنگام اجرای مجدد IDA، پنجره IDA Log، پیام هایی به صورت زیر تولید خواهد کرد:

```
[*] pStalker> Process Stalker - Profiler
[*] pStalker> Pedram Amini XXXXXXXXXXXXXXXXXXXXXX
[*] pStalker> Compiled on Jul 5 2005
```

پلاگین IDA برای باینری های پیش-پردازشی[6] مورد استفاده قرار گرفته و خروجی های زیر را تولید خواهد کرد:

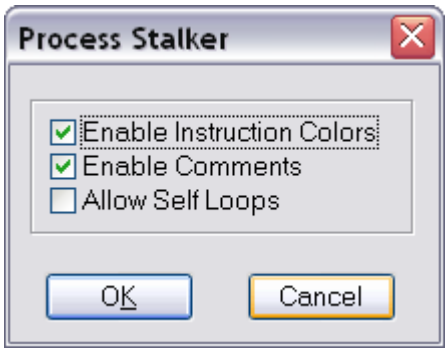
- گراف های سطح-تابع و واحدِ GML

---

[1] run-time profiling
[2] state mapping & tracing
[3] run-time block-level trace
[4] tracing
[5] instrumenting
[6] pre-processing

- فایل Cross References، که تمام فراخوانی های توابع-متقابل را لیست خواهد کرد.
- لیست Break Point، با مقادیری برای هر یک از بلاک پایه موجود در باینری.

به منظور اجرای پلاگین، منتظر پایانِ آنالیز اتوماتیک IDA باشید یا آنرا با کلیدهای Alt+5 اجرا کنید. همچنین می توانید با استفاده از منوی Edit و انتخاب گزینه plugins این کار را نیز انجام دهید. پنجره زیر بایستی ظاهر شود:



Process Stalker IDA plug-in dialog

گزینه های پیش فرض در بسیاری از موارد بایستی کافی باشند. به هر حال در زیر توضیحی راجع به هر یک از این گزینه ها داده می شود:

- **Enable Instruction Colors**: کنترل می کند که آیا پلاگین باید دستورها را به صورت رنگی تولید کند یا خیر. هنگامی که با گراف های بزرگ کار می کنید، غیرفعال کردن رنگ بندی در سطح-دستور می تواند به افزایش عملکرد پلاگین کمک کند.
- **Enable Comments**: دخالت comment های IDA را در گراف های تولید شده فعال یا غیرفعال می کند.
- **Allow Self Loops**: طرح بندی قائم در گراف هایی که حاوی self loop ها باشند، موجود نخواهد بود.

هنگامی که گزینه های مطلوب انتخاب شدند، OK را فشار دهید و یک شاخه را برای قرار دادن فایل های خروجی تعیین کنید. راجع به تعیین گزینه filename نگران نباشید. نام فایل به صورت اتوماتیک تولید خواهد شد. پلاگین هنگام آنالیز کردن باینری مقداری زمان نیاز داشته و سپس پیام های زیر را در پنجره IDA Log ظاهر خواهد کرد:

```
[*] pStalker> Profile analysis 25% complete.
[*] pStalker> Profile analysis 50% complete.
[*] pStalker> Profile analysis 75% complete.
[*] pStalker> Profile analysis 100% complete.
```

گراف ها به زبان GML تولید می شوند که به نرم افزار **GDE Community Edition** تعلق دارد. نرم افزار مربوطه، یک نرم افزار رایگانِ GML Viewer بوده که از وبسایت www.oreas.com قابل دانلود می باشد.

گراف ها، تعاملی و قابل ویرایش می باشند و رنگ بندی مناسب سطح-دستور دارد که برای خوانایی بهتر کاربرد داشته و می توان آنها را با تعدادی الگوریتم طرح بندی از قبیل سلسله مراتبی (hierarchical)، قائم (orthogonal)، متقارن (symmetric) و مدور (circular) و ... نمایش داد. نقاط ثبت[7] با رنگ سبز، نمایش داده شده، شاخه های[8] درست/غلط به تناسب به صورت سبز/قرمز رنگ بندی شده و لبه های مجازی به رنگ آبی در می آیند.

[7] Entry points

Process Stalker – ردیاب

ابزار مستقل ردیابی که متعلق به Process Stalker می باشد، احتیاج به هیچ عملیات نصب و ... ندارد. فایل مستقل process_stalker.exe، یک ابزار command-line بوده که برای بارگذاری یک پروسه یا ضمیمه شدن[9] به آن استفاده می شود.

```
usage:
  process_stalker <-a pid | -l filename | -la filename args>

options:
  [-b bp list]    specify the breakpoint list for the main module.
  [-r recorder]   enter a recorder (0-9) from trace initiation.
  [--one-time]    disable breakpoint restoration.
  [--no-regs]     disable register enumeration / dereferencing.
```

- **-a pid** : به پروسه ای که مشخصه پروسه آن برابر pid می باشد، ضمیمه می شود.
- **-l filename** : پروسه را بدون هیچ گزینه ای از command-line بارگذاری می کند.
- ***-la filename args***: یک پروسه را بارگذاری کرده و آنرا به آرگومان هایی که توسط args مشخص شده اند، انتقال می دهد (pass).
- ***-b breakpoint list***: لیست **breakpoint** تعیین شده را برای **image** اصلی، بارگذاری و تجزیه می کند. به خاطر داشته باشید که احتیاجی به تعیین این آرگومان برای **DLL** هایی که به طور متوالی بارگذاری شداند، ندارید.
- ***-r recorder***: بلافاصله، عملیات ضبط را در recorder تعیین شده، شروع می کند. آرگومان های معتبر، 0 تا 9 هستند. خروجی ضبط شده در [pid | filename].[recorder #] ذخیره می گردد.
- **--one-time** : اگر این گزینه مشخص شده باشد، بازیابی های breakpoint را غیرفعال می سازد. مشخص نمودن این گزینه موجب افزایش معنی دار عملکرد پلاگین می شود.
- **--no-regs** : اگر این گزینه مشخص شده باشد، عملیات enumeration و dereferencing را روی ثبات غیر فعال می سازد. مشخص نمودن این گزینه موجب افزایش معنی دار عملکرد پلاگین می شود. خروجی ضبط برای ثبات، در [pid | filename]-regs.[recorder #] ذخیره می گردد.

لیست های breakpoint مربوطه بایستی در شاخه ای که process stalker از آنجا اجرا می شود، وجود داشته باشند. لازم است یک لیست breakpoint اولیه، بر روی خط فرمان فقط برای تصویر اولیه و اصلی تعریف شود. به محض اینکه پروسه مورد نظر ماژول ها را بارگذاری کرد، Process Stalker شاخه فعلی را برای وجودِ MODULE_NAME.bpl بررسی می کند. اگر فایل ذکر شده وجود داشت، لیست breakpoint به صورت اتوماتیک بارگذاری و تجزیه خواهد شد. خروجی زیر شروعِ یک نشست موفقیت آمیز از Process Stalker را روی PID 864 که همان Hyper Terminal است، نشان می دهد:

```
$ process_stalker -a 864 --one-time

process stalker
pedram amini XXXXXXXXXXXXXXXXXXXXXX
compiled on Jul 5 2005
target: 864
```

---

[8] Branch
[9] attach

```
processing breakpoints for module HYPERTRM.dll at 67441000
done. 10568 of 10577 breakpoints set.

initial break, tid = 0794.

commands: [h]   this screen                    [m] module list
          [0-9] enter recorder modes           [x] stop recording
          [v]   toggle verbosity
          [d]   detach (XP/2003 only)          [q] quit/close

004d68b2 T:00000a74 [bp] 6747D041 mov edi,edi
004d68b2 T:00000a74 [bp] 6747D05C cmp [ebp+0C],0x1
004d68b2 T:00000a74 [bp] 6747D0AB cmp [ebp+0C],esi
004d68bc T:00000a74 [bp] 6747D0EB xor eax,eax
...
```

به محض اینکه به breakpoint ها برخورد می شود، Process Stalker یک مقدار log را در STDOUT ایجاد می کند. قالب این مقدار به این صورت می باشد: اولین فیلد، **004d68bc**، علامت شمارشی فعلی را به محض اینکه توسط ویندوز گزارش شد، نمایش می دهد. فیلد دوم، **T:00000a74**، مشخصه رشته ای[10] را که این breakpoint به آن دست پیدا کرده است، نمایش می دهد. سومین فیلد، **[bp]**، recorder ای را نشان خواهد داد که این entry در آن نوشته شده است. عملیات ضبط کردن در حالت پیش فرض، خاموش (off) می باشد و توسط تگ [bp] مشخص می شود، در غیر این صورت شماره ضبط (recording number)، در تگ ظاهر می شود. فیلد چهارم، **6747D0EB**، آدرس breakpoint را نشان می دهد. آخرین فیلد، **xor eax,eax**، دیزاسمبلیِ دستورات در آدرس **breakpoint** را نشان می دهد.

HotKey ها را می توان برای کنترل Process Stalker در هر لحظه ای از اجرا به کار برد. کلیدهای شماره ای، **[0-9]**، به Process Stalker علامت خواهد داد که باید عملیات ضبط را در شکافِ مشخص شده ی recorder، شروع کند. نام فایلِ recorder توسط pid ضمیمه شده یا filename بارگذاری شده مشخص می گردد و با پسوندِ [recorder #] اضافه می گردد. فایل های Recorder الحاق شده اند و قابلیت جای نویسی(overwrite) ندارد. برای بستن یک recorder، کلید [x] را فشار دهید. کلید [v]، لاگ سازی خروجی در سطح-breakpoint را کنترل می کند. عملیات لاگ سازی را برای افزایش کارآیی غیرفعال سازید. کلید [m]، لیستی از تمام ماژول های بارگذاری شده و لیستی از ماژول های در حال پایش را نمایش خواهد داد.

```
---------- MODULE LIST ----------
module 01001000 HYPERTRM.EXE
module 7c801000 KERNEL32.dll
module 7c901000 ntdll.dll
module 7c9c1000 SHELL32.dll
...

stalking:
67441000 - 67480000 [0003f000] HYPERTRM.dll
---------- MODULE LIST ----------
```

کلید [q] را برای بستن هر عملیات recording که فعال بوده (یا باز می باشد) و نیز بستن debugger ها به کار برید. همچنین، روی سیستم هایی که Detaching یا جدا سازی را پشتیبانی می کنند (Windows XP و 2003) از کلید [d] برای بستن هر عملیات recording باز، و نیز reset کردن تمام breakpoint ها و جدا ساختن از debugger استفاده کنید.

[10] Thread ID

ابزارهای پیتون

تعدادی از ابزارهای Python، برای سنجش لیست های breakpoint، recording ها، آمارها و گراف ها ارائه شده اند، به خاطر داشته باشید که، ابزاری که با recording ها کار می کنند، انتظارِ recording های پردازش شده را می کشند (ps_process_recordings) را ببینید). توضیح و مثالی کاربردی از هر ابزار را در زیر (که به صورت الفبایی چیدمان شده اند) می بینید:

ps_add_register_metadata.py

Process Stalker Graph Register Metadata Importer

این ابزار برای اضافه کردن metadata ی ضبط شده از ثبات (recorded register metadata) به گراف تعیین شده ارائه شده است. مثال:

```
ps_add_register_metadata <recording-regs> <GML file> [rebase address]

ps_add_register_metadata 2360-regs.0 in.gml > out.gml
```

گراف خروجی با داده های زنده (live) در ثبات ترجمه (render) می شود و مقادیر آنیِ ۳۲-بیتی ثبات را نشان می دهد. ثبات هایی که به صورت اشاره گرهای نهانی بدرون فضای stackیا heap تعیین شده اند مجددا ارجاع زدایی (dereference) شده و نمایش داده می شوند. اگر یک رشته ASCII یا UNICODE تشخیص داده شود، به جای داده های ۴-بیتی ردیف شده (aligned)، نمایش داده می شود.

ps_bp_filter.py

Process Stalker Breakpoint Filter

این ابزار به عنوان یک وسیله برای فیلتر کردن لیست های breakpoint تولید شده ارائه شده است.

- تابع – breakpoint هایی که در یک تابع نیستند فیلتر می کند. یعنی دیگر احتیاجی نیست که به جزئیات سطح-بلوک[11] توجهی کنیم.
- لیست تابع – breakpoint هایی که به هر تابع تعلق دارد را فیلترمی کند. با این تفاوت که توابع یاد شده توسط یک لیست تابعی که توسط کاربر ارائه شده اند مشخص می گردند.

تعدیل کننده[12] ی "in/out" تعیین می کند که آیا توابع موجود در لیست تابع باید به لیست خروجی فیلتر شوند یا به بیرون لیست ورودی. خروجی این فایل را باید به یک لیست breakpoint جدید تغییر مسیر داد (یا به اصطلاح Redirect کرد). مثال کاربردی:

[11] block-level

```
ps_bp_filter <in bpl> <out bpl> <functions|<func1>:[func2]:[...] <in |
out>>
```

```
ps_bp_filter input.bpl filtered.bpl functions
ps_bp_filter input.bpl filtered.bpl 00001234:deadbeef in
```

این ابزار، خیلی اوقات برای میزان سازی خوبی از Process Stalking ما، تنها روی نواحی دلخواه و جالب از کد استفاده می شود. موردی را فرض کنید که یک پروسه را با یک GUI، stalk می کنید. یک عملیات recording را در حین اینکه به شدت با GUI در حال تقابل و interact هستیم، روی پروسه هدف انجام می دهیم. از ابزار ps_recording_to_list با یک آرگومان 'module' برای استخراج لیست تابع از recording استفاده کنید. از ps_bp_filter برای حذف این گره های غیرمطلوب از لیست breakpoint استفاده کنید.

ps_gde_fixup.py

Process Stalker GDE-Generated GML Fixer

Oreas GDE گاهی اوقات فایل های GML ی را که نتوانسته به درستی تفسیر (parse) کند ذخیره و save می کند. هنگامی که به این مورد برخوردید از این اسکریپت برای ترجمه مجدد (re-render) و تفسیرِ GML Description خراب شده استفاده کنید. مثال کاربردی:

```
ps_gde_fixup <in.gml>
```

```
ps_gde_fixup gde-generated.gml
```

ps_graph_cat.py

Process Stalker Graph Concatenate

این ابزار برای ترکیب گرافِ چندین تابع به یک گراف بزرگتر یا (به صورت اختیاری) به یک گراف بهم پیوسته[13] ارائه شده است. گره ها توسط تابع با یکدیگر به صورت خوشه ای در می آیند (و یک جا گرد هم می آیند). مثال کاربردی:

```
ps_graph_cat [-x <xrefs file> -b <base addr>] <file_1> [file_2] ...
```

```
ps_graph_cat.py -x dll.xrefs -b 46001000 1.gml 2.gml > out.gml
```

نام فایلِ cross-references و آدرس پایه به طور متوالی از متغیرها استفاده می کنند، بنابراین این ابزار به شما کمک می کند که آنها را در محیط اولیه خود اضافه کرده و کار خود را راه اندازی کنید. به خاطر داشته باشید که یک کاهش مهم در کاراییِ تجسم سازی و تقابل اثر با گراف های بزرگ بهم پیوسته وجود دارد.

---

[12] Modifier
[13] Interconnected Graph

ps_graph_highlight.py
Process Stalker Graph Highlighter

این ابزار برای برجسته سازی گره های بالقوه جالبِ گراف استفاده می شود. گره های که حاوی هر یک از موارد زیـر باشـند را گره های جالب تعریف می کنیم:

- حلقه های Instruction Level (REP MOVS و ...).
- فراخوانی به API ای که روی رشته دستکاری می کند[۱۴] (wstrcpy ، strcat ، sprintf و ...).
- فراخوانی به API ای که تخصیص و دستکاری حافظه را انجام می دهد[۱۵] (malloc، LocalAlloc و ...).
- دستورهای انفصالی[۱۶] (INT 80 و ...).

گزینه های command line برای کنترل اینکه آیا باید گره های بالقوه جالبِ "all" یا "hit" بایستی برجسته سـازی شـوند یـا خیر. همچنین برای کنترل شرایط و وضعیت هـای برجـسته سـازی (reps، ints، str، mem و ...) در اختیـار قـرار گرفتـه انـد. مثـال کاربردی:

```
ps_graph_highlight [--nodes hit,missed,all] [--reps,--ints,--str,--mem] <GML>
ps_graph_highlight --nodes hit in.gml              > graph-highlighted.gml
ps_graph_highlight --nodes all --reps --str in.gml > graph-highlighted.gml
```

ps_idc_gen.py
Process Stalker IDC Script Generator

این ابزار یک فایل GML را تفسیر کرده و تمام نام های توابع و comment های کاربری را شمارش کرده یا enumerate می کند. یک اسکریپت IDC تولید خواهد شد که می توان آن اسکریپت را برای import کردن تغییرات به بانـک اطلاعـاتی IDA، به کار برد؛ که این تغییرات در حقیقت تغییراتی هستند که مستقیما درون ویرایشگرِ گراف[۱۷] انجام شده اند. مثال کاربردی:

```
ps_idc_gen <file 1> [file 2] [...]

ps_idc_gen.py *.gml > import_changes.idc
```

ps_process_recording.py
Process Stalker Recording Post Processor

این ابزار برای پردازشِ یک عملیات recording ارائه شده اند تا با پردازشِ یک عملیـات recording، مـا بتـوانیم offset های تابع را در آدرس های breakpoint اضافه کنیم و به صورت اختیاری آدرس های recording را **rebase** کنیم. مثال کاربردی:

```
ps_process_recording <recording> [base address]

ps_process_recording recording.0
```

---

[14] string manipulation API
[15] memory allocation and manipulation API
[16] Interupt instructions
[17] Graph Editor

دستور فوق برای هر رشته مواجهه که recording را پیدا کند، یک فایل مجزا تولید خواهد کرد. FileName ها به صورت زیر تولید می شوند:

recording.0.thread_id-processed

این ابزار انتظار دارد که لیست های breakpoint ِ متناظرِ bpl. در شاخه فعلی وجود داشته باشند.

ps_recording_to_list.py
Process Stalker Recording-Function List Generator

این ابزار به عنوان یک راهنما و کمکیار برای پردازش یک recording، بدرون یک لیست تابع (که بعدا می توان این لیست تابع را در یک فراخوانی به ابزار فیلتر breakpoint[18] به کار برد) ارائه شده است. این ابزار به صورت اختیاری یک گزینه ی دوم را برای command-line می گیرد که با استفاده از آن بتوان، تنها توابعی که به یک ماژول مشخص تعلق دارند فیلتر کرد. مثال کاربردی:

```
ps_recording_to_list <processed recording> [module]

ps_recording_to_list recording.0.processed
```

در صورتی که خروجی این ابزار را می خواهید به عنوان یک آرگومان برای ps_bp_filter بکار برید، باید آرگومان اختیاریِ 'module' را مورد استفاده قرار دهید.

ps_recursive_view.py
Process Stalker Recursive Graph Viewer

این ابزار برای ترکیب چندین گرافِ تابع به یک گراف بزرگتر و بهم پیوسته ارائه شده است. با یک گراف (تابع) معین، این ابزار به صورت بازگشتی تمامی فراخوانی های تابعیِ inter-module را شمارش کرده یا enumerate می کند. توابع با هم بصورت خوشه ای در می آیند و در یک جا در کنار هم گرد می آیند. مثال کاربردی:

```
ps_recursive_view <GML file> <xrefs file> <base address>

ps_recursive_view graph.gml xrefs DEADBEEF > out.gml
```

همان اخطار بالایی نیز در اینجا باید تذکر داده شود، گراف های بزرگ و بهم پیوسته بسیار دیر ترجمه و render می شوند. اگر شما احتیاج به تصویر سازی از گراف های بزرگ دارید، از یک الگوریتم طرح بندی سریع تر مثل دایره ای استفاده کنید که بر خلاف سلسله مراتبی یا قائم دارای سرعت بالایی در عملیات rendering است.

ps_state_mapper.py
Process Stalker State Mapper

[18] Breakpoint Filter utility

این ابزار یک "وضعیت نگاشت" یا "state mapping" را از چندین فایلِ recorder می سازد. تصویرسازی به صورت سطح-تابع بوده و بوسیله وضعیت recorder (recorder stat)، گره ها را خوشه ای کرده و گرد هم می آورد. مثال کاربردی:

```
ps_state_mapper <xrefs file> <base address> <recorder0 recorder1
[...]>

ps_state_mapper module.xrefs DEADBEEF recorder.0 recorder.1 > out.gml
```

یک استفاده جالب از این تصویرسازی برای نمودار کشیدنِ کمالِ یک fuzzer در حال توسعه است. یک نمایشِ stat mapper را می توان برای مثال روی یک پایه هفتگی تولید کرد و آنها را مورد مقایسه قرار داد. این مقایسه می توان دید که آیا fuzzer در فضای پردازشی مقصد دقیق تر و عمیق تر (deeper) می شود یا خیر.

ps_view_recording_funcs.py

Process Stalker Recording Function Viewer

این ابزار به عنوان یک وسیله ای برای تصویر سازی توابعی که درون یک عملیات recording، "hit" شده اند، ارائه شده است. هر بلوک پایه برای هر تابعِ hit نمایش داده خواهد شد. بلوک های پایه ای که "hit" شده اند، برجسته سازی خواهند شد. مثال کاربردی:

```
ps_view_recording_funcs [-f function] [-x <xrefs file> -b <base addr>]
<recording>

ps_view_recording_funcs recording.0.processed             > out.gml
ps_view_recording_funcs recording.0.processed -f deadbeef > out.gml
```

آرگومان اختیاری 'function' را برای نمایش hit-graphِ فقط همان تابع تعیین کنید. به خاطر داشته باشید که این ابزار انتظار دارد که گراف های تابعیِ gml. در شاخه فعلی وجود داشته باشند.

ps_view_recording_stats.py

Process Stalker Recording Statistics

این ابزار، آمارهای پایه ای روی یک عملیات recording را تولید خواهد کرد. این آمارها شامل شمارش hit به ازای هر تابع[19] و زمان انتقالی تابع به تابع[20] می باشند. مثال کاربردی:

```
ps_view_recording_stats <recording> [sort]

ps_view_recording_stats recording.0.processed
ps_view_recording_stats recording.0.processed sort
```

آرگومان اختیاری 'sort' را برای مرتب کردن (دسته کردن) خروجی براساس زمان/شمارش در مقابل آدرس تعیین کنید.

---

[19] per function hit count
[20] function to function transition time

ps_view_recording_trace.py

Process Stalker Recording Trace Viewer

این ابزار به عنوان وسیله ای برای تصویر سازی از نتایج یک فایل recorder به صورت ترتیب متوالی[۲۱] (یا پی در پی) ارائه شده است. این ابزار توالی های تکراری[۲۲] را **fold** می کند (یا به اصطلاح تا می زند) و آنها را با یک شمارش تکراری[۲۳] گرد هم می آورد (خوشه ای می کند). مثال کاربردی:

```
ps_view_recording_trace <recording>

ps_view_recording_trace recording.0.processed > out.gml
```

گراف تولید شده برای تعیین محل کردن حلقه های زمان-اجرا و منطقیِ گردشی (یا گردنده)[۲۴] مفید می باشد. به خاطر داشته باشید، این ابزار انتظار دارد که گراف های تابعیِ gml. در شاخه فعلی وجود داشته باشند.

API ابزاریِ پیتون

مجموعه ابزار پیتون برای اکثر کاربران عاملیت ابزاری کافی را ارائه خواهد داد. برای کاهش احتیاجات کاربران قدرتمندتر، یک API، برای خلاصه کردنِ عملیات خسته کننده ای مثل تفسیر (یا تجزیه) فایل (file parsing) و ترجمه گراف (graph rendering) توسعه یافت. این API به صورت inline بوسیله Epydoc سند سازی شده و ماژول ها/کلاس های زیر را استخراج یا export خواهد کرد:

- **gml**
- gml_node
- gml_edge
- gml_cluster
- gml_graph
- **ps_parsers**
- bpl_parser
- recording_parser
- register_metadata_parser
- xrefs_parser

تمام مجموعه ابزار پیتون، علیه این API ساخته شده اند و مثال های کاربردی خوبی برای شروع مناسب هستند. با استفاده از این ابزار به عنوان نقطه شروع کار، می توانید ابزارهای جدیدی تولید کرده و توسعه دهید. سندسازی ها با جزئیات بیشتر را می توانید در شاخه 'ps_api_docs' مشاهده کنید:

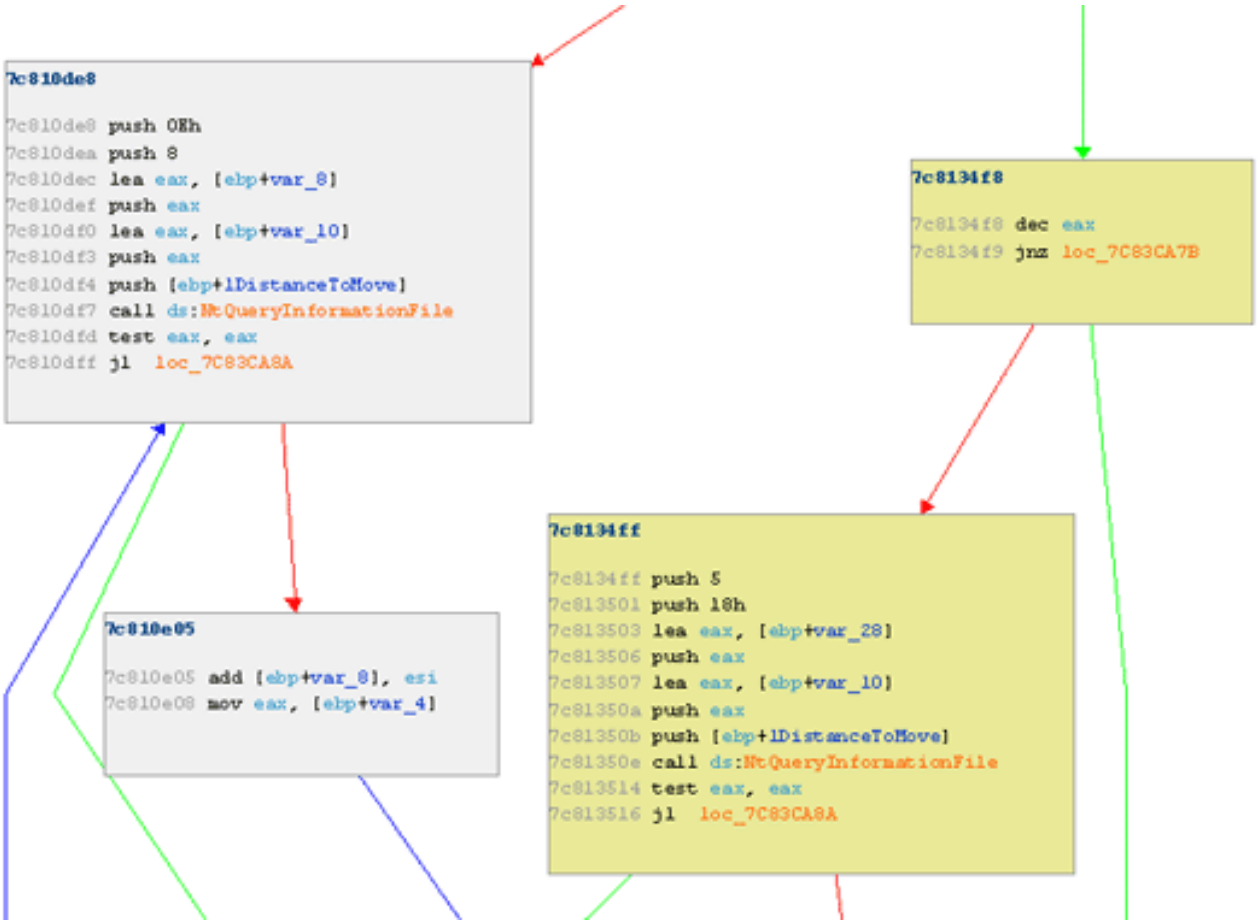
Epydoc Generated API Reference:
http://www.idefense.com/iia/releases/ps_docs/ps_api_docs/index.html

---

[21] sequential order
[22] repeat sequences
[23] repeat count
[24] locating run-time logic driven loops

گراف های نمونه

با استفاده از ترکیب filtering و تجهیزاتِ گرافی، تصویرسازی های بیشتری را می توان با ابزار توضیح داده شده در فوق ایجاد کرد. برای آشنا کردن کاربران با موضوع، مثال ها و عکس هایی از گراف های برگزیده کامپایل شده اند. گراف برگزیده زیر، گنجایش (inclusion) و برجسته سازی (highlighting) به کدِ تابعِ تکه شده یا "chunked"[۲۵] را نشان می دهد:



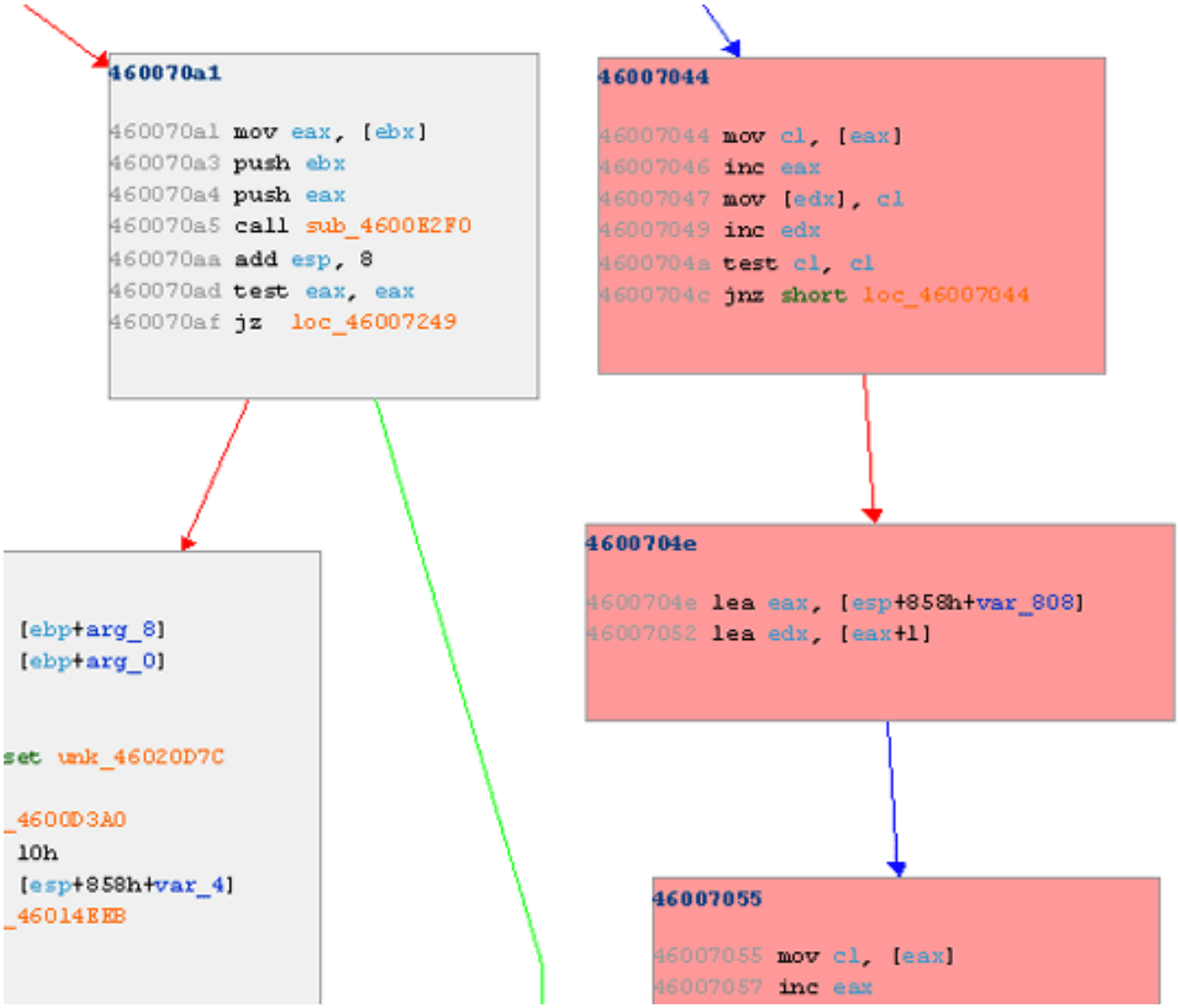
گره هایی که با رنگ زرد می باشند، نمایانگر قطعه های chunked می باشند.

توابع chunked اساسا در باینری های بهینه سازیِ کامپایل شده از ماکروسافت دیده می شوند. در زمان کامپایل، کامپایلر ماکروسافت کد مشابه و غیرمحتمل بر اجرا[۲۶] را ترکیب کرده و آنرا از توابع گوناگون استخراج می کند و آنها را در خارج از فضای تابعی مجاور (contiguous function space) نگه داری می کند. قطعه های تابع در نوار Navigation در IDA، با رنگ قهوه ای نمایش می یابند. گرافر استاندارد IDA ، تصویر سازی از قطعات تابع را پشتیبانی نمی کند و آنها را به سادگی به صورت بلوک های قرمز رنگ و بی فایده نمایش می دهد. با آمدن IDA 4.7 پشتیبانی از تابع های chunked در SDK پشتیبانی می شود، اما گرافر (grapher) یا تصویرساز (عاملی که وظیفه تولید یا آماده سازی تصویر را بر عهده دارد) از آن استفاده نمی کند. برای مهندس امنیت، کد غیر محتمل بر اجرا، جالب است، مخصوصا زمانی که روتین های بازرسی (یا بررسی) امنیتیِ معمول به این صورت دسته بندی (یا طبقه بندی) شده باشند. رشته ی Process Stalking، هم در ردیابی (یا tracing) و هم در تصویرسازی (visualization) از توابع قطعه ای یا chunked پشتیبانی می کند.

گراف زیر نکات برجسته سطح انسداد پایه را نشان می دهد و می توان پس از یک ردیابی زمان-اجرا با استفاده از ابزاری چون ps_process_recording و ps_view_recording_funcs آنرا ساخت.

---

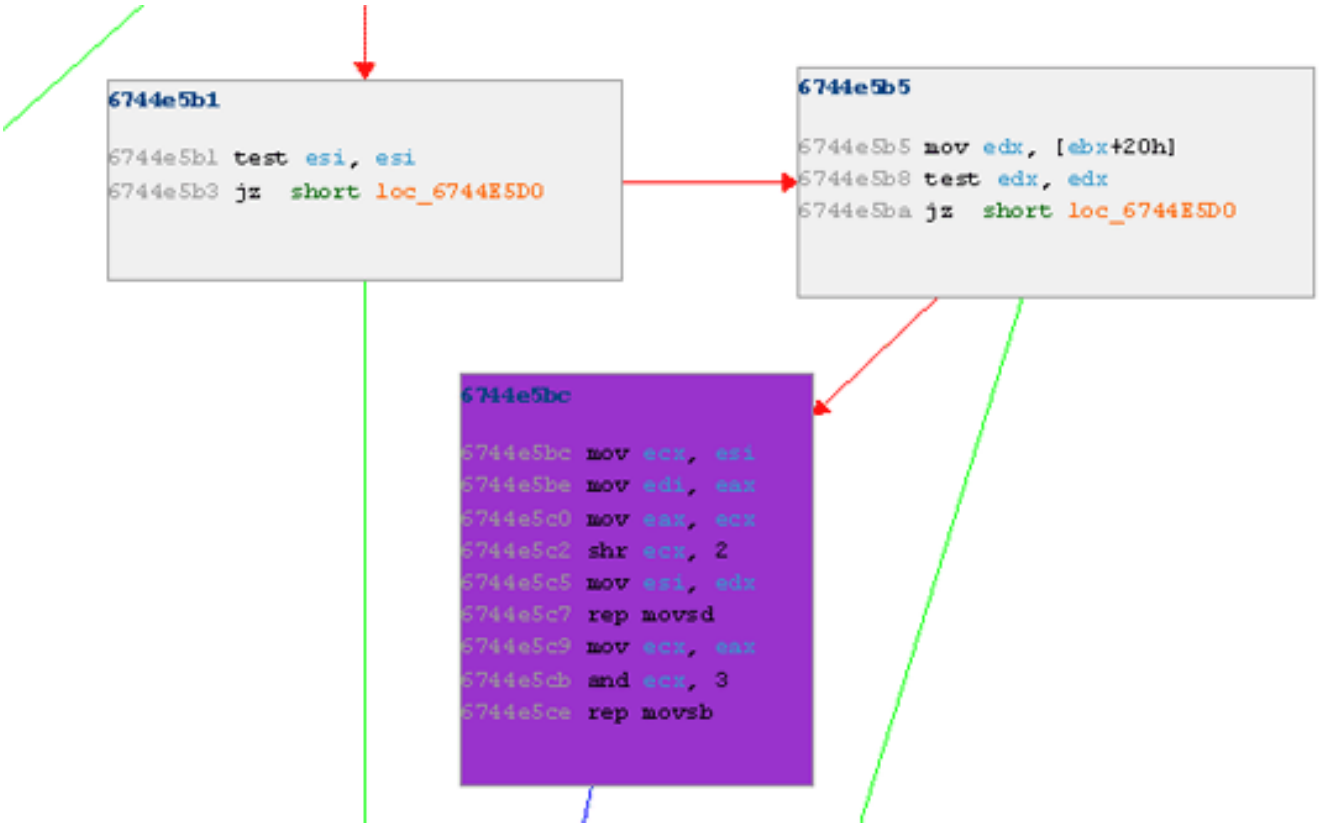
[25] "chunked" function code
[26] "less likely to be executed"



Nodes highlighted in red represent run-time hits.

یک مشکل معمولا در زمان هدایت یک بازرسی از کد سطح اسمبلی بوجود می آید، تعیین مکان شروع بازنگری است. بسیاری از باینری ها حاوی تعدادی توابع هستند که از دامنه کم انتها (500) تا دامنه های پر انتها (بیش از چند هزار) امتداد دارند. محققان معمولا با استفاده از روتین های کتابخانه ای شناخته شده استفاده به عمل می آورند تا ورودی کاربر را فرآیند کنند (از قبیل روتین recv()، و به طور دستی از طریق کدهای traversed (متقاطع) آنرا ردیابی می کنند. با استفاد ه از رشته Process Stalking، ما قادریم که به سرعت و به صورت بصری تعیین کنیم که کدام توابع در فرآیند ورودی های مشخصی شرکت دارند و همچنین می توان تعیین کرد که کدام یک از انسدادهای پایه موجود در آن تابع، متقاطع (traversed) شده اند.

باید توجه داشت که تجسمِ فرآیند ردیابی تولید شده می تواند زمان بر باشد. در حالی که ابزار ps_graph_highlight که سریع تر از روش های سنتی است، را می توان با قاطعیت جهت تمایز بلوک هایی از کد بالقوه جالب به کار برد.



Nodes highlighted in purple represent potentially "interesting" nodes.

ابزار ps_graph_highlight را می توان از طریق گزینه های command line، بسادگی توسعه پذیر، قابل کنترل ساخت. آنالیست می تواند گزینه های زیر را برجسته (highlight) نماید:

1. **گره های بالقوه جالب**
2. **گره های بالقوه جالبی که از طریق فرآیند ردیابی traverse شده اند.**
3. **گره های بالقوه جالب که از طریق فرآیند ردیابی traverse نشده اند.**

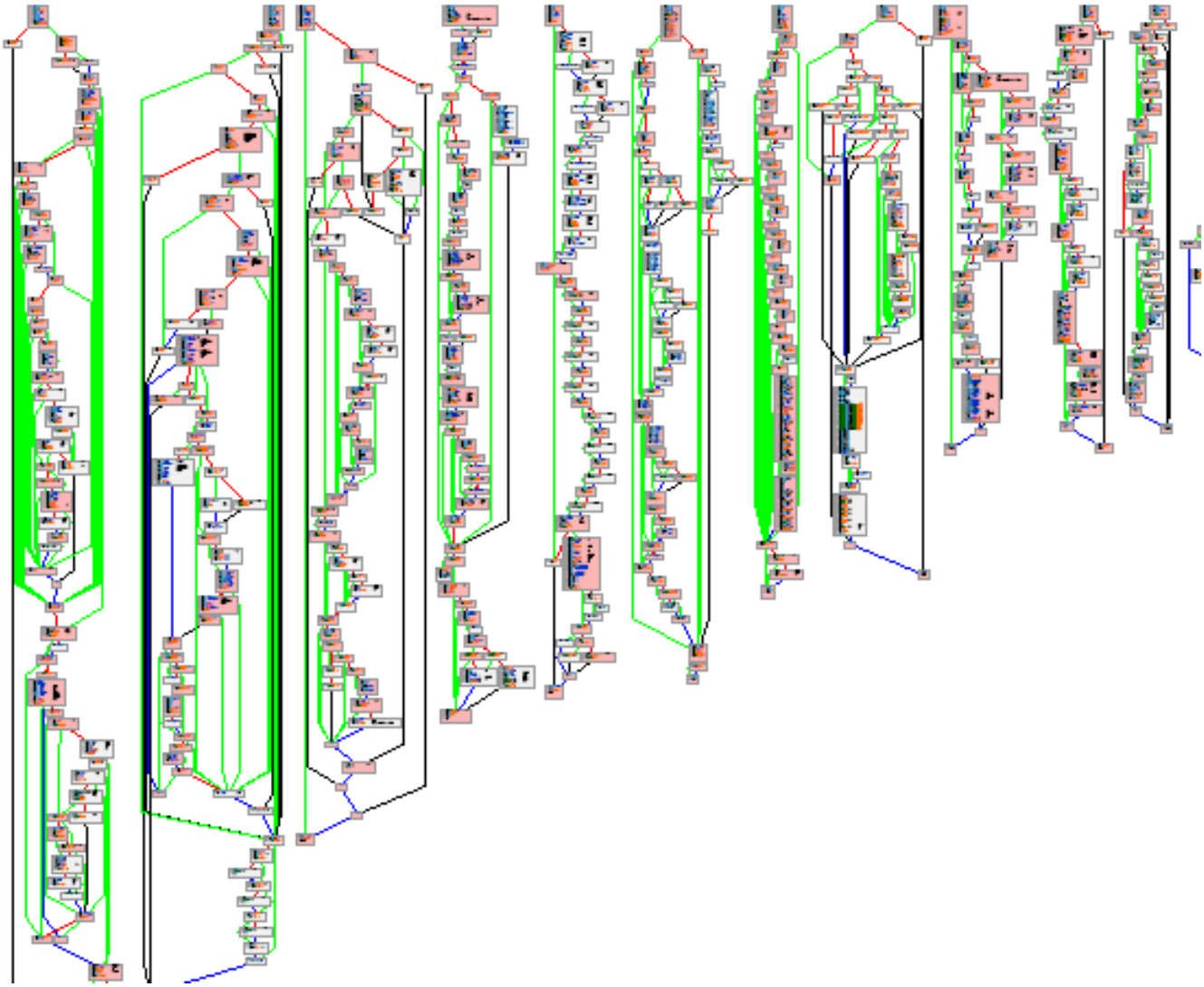
در گراف فوق بلافاصله یک کپی رشته ایِ inline تشخیص داده شده و highlight شده است. سه گره پیشین موجود در گراف، شرایط شاخه را می توان تجزیه و تحلیل نمود و چگونگی تغییر و اصلاح داده های کاربری ارائه شده را (به منظور تغییر روند کنترل جهت رسیدن به این بلوک بالقوه آسیب پذیر) تعیین نمود.

Disassembly ایستای برجسته شده به محققان اجازه می دهد که بلافاصله مسیرهای اجرایی کد را ببینند. برای تعیینِ مناسب نوع داده هایی که بایستی از طریق گره های خاص، مورد بررسی قرار گیرند، Process Stalker را می توان به گونه ای تنظیم نمود (هدایت کرد) که metadata های ثبات ها را ذخیره و تفسیر کند.

Register values are displayed with automated "smart" dereferencing and string detection.

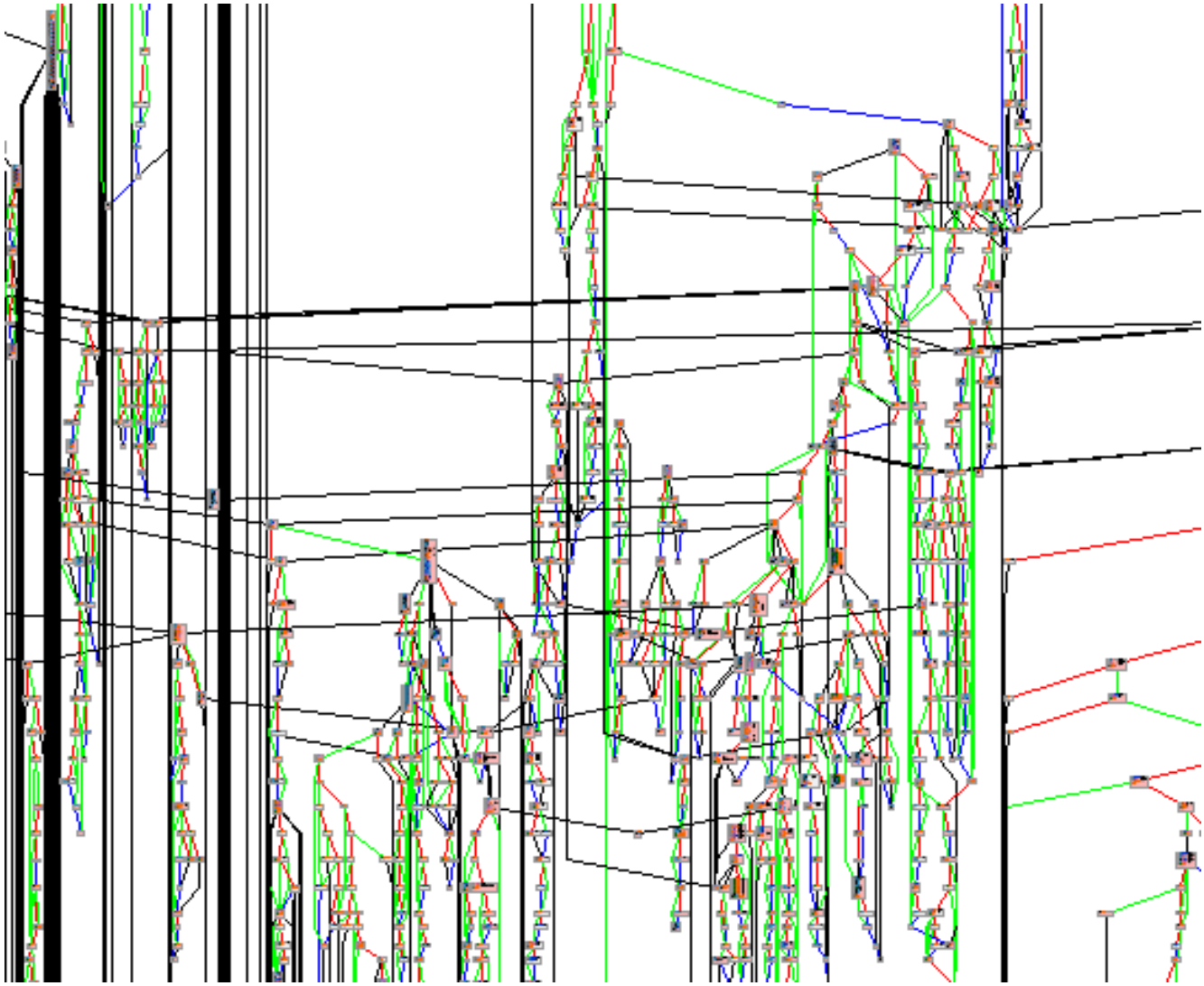
گره ها از گراف فوق با داده های زنده ای از ثبات، تفسیر شده اند و مقادیر ۳۲ بیتی ثبات ها را به طور همزمان نشان می دهد. ثبات هایی که به عنوان اشاره گرهای بالقوه در فضای پشته یا heap تعیین شده اند، مرجع زدایی شده (dereferenced) و نشان داده می شوند. اگر یک رشته ASCII یا UNICODE تشخیص داده شود، آنگاه می تواند بجای داده های ۴ بیتی بصف شده (aligned) نمایش داده می شود.

تعدادی از الگوریتم های طرح بندی در اختیار کاربران وجود دارد، که هر کدام مزیت خود را دارد. در گراف زیر، تعدادی از توابع در یک گراف واحد با طرح بندی سلسله مراتبی نشان داده شده است:



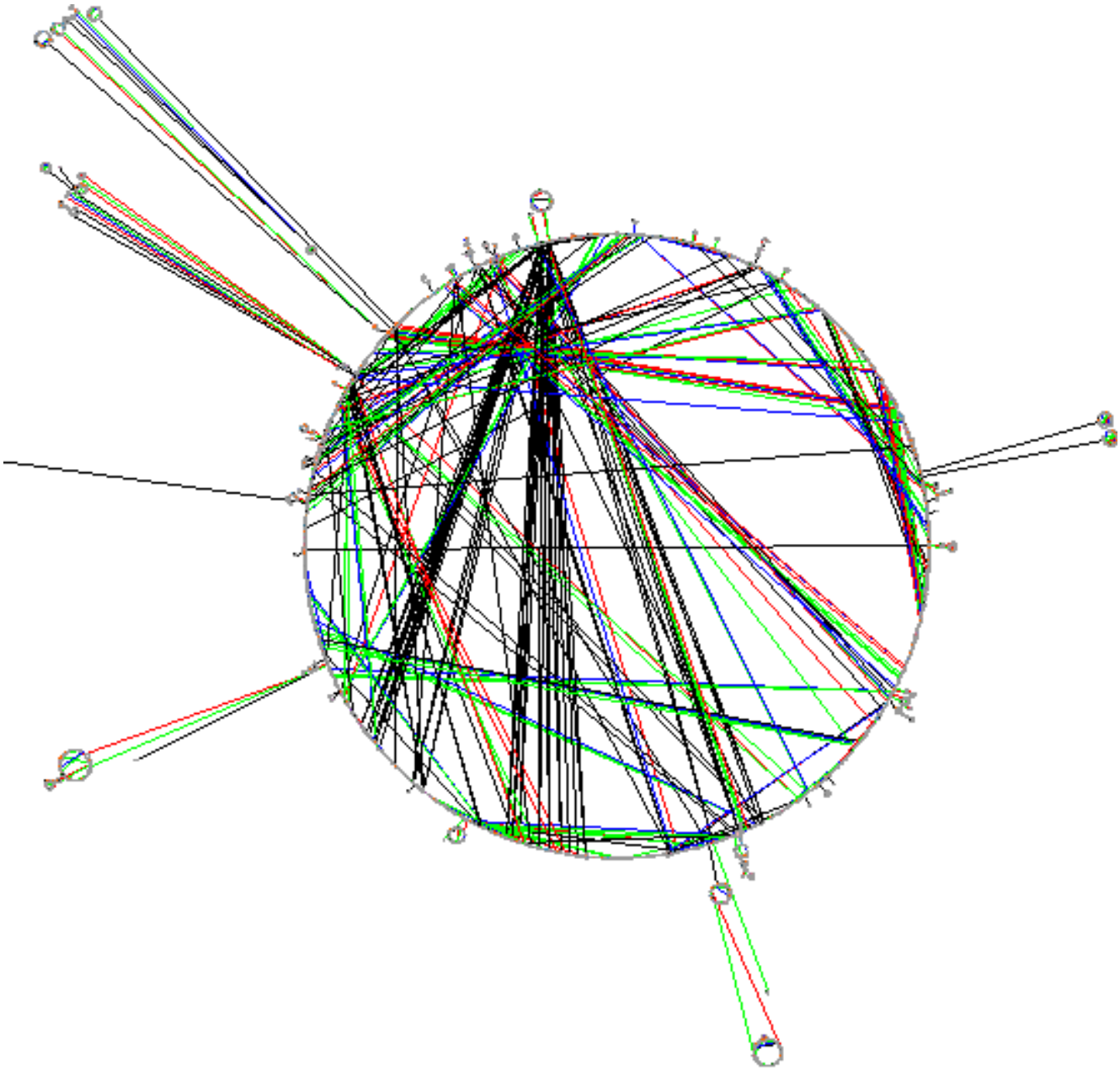
Low-res excerpt from graph concatenation of multiple graphs.

گراف فوق را می توان به وسیله ps_view_recording_funcs یا ps_graph_cat تولید کرد. مجددا خاطر نشان می شود که تعداد زیادی از روش ها در اختیار محققین وجود دارد که قابلیت انعطاف زیادی دارند. همان طور که در بالا ذکر شد، گره های قرمز نشان دهنده بلوک های پایه هستند که در فرآیند ردیابی با آن برخورد می کنیم. این گراف خود از یک گراف بسیار بزرگ تری گرفته شده است که دارای حداقل ۲۰ تابع می باشد. گرچه توصیه نمی شود که یک چنین الحاق های بزرگی از گراف ها را برای اتصالات متقابل بکار برد، مع الوصف محقق می تواند این کار را انجام دهد.



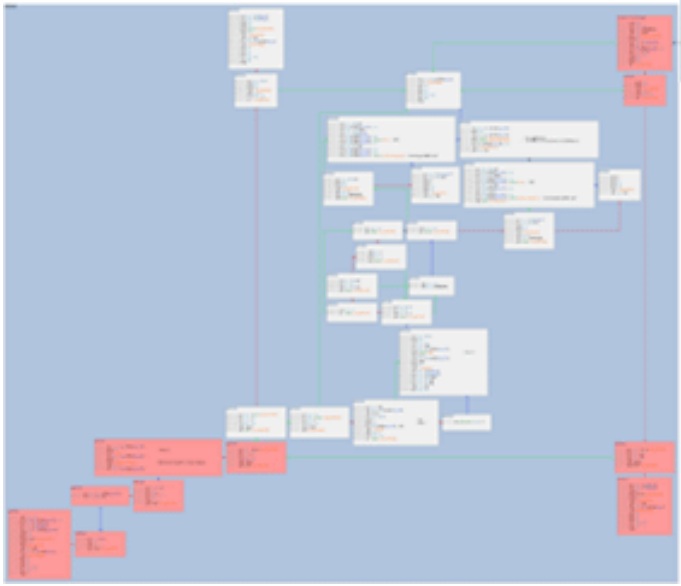
Low-res excerpt from interconnected graph concatenation of the same multiple graphs from above.

گراف تولید شده ی فوق بسیار پیچیده است و فقط به عنوان یک کار نمایشی، ارائه شده است. لازم است که برای کاهش دادن مجموعه توابع مورد نیاز برای بصری کردن (تجسم سازی) از فیلترینگ های بیشتری استفاده کرد. همچنین در مواردی که استفاده از یک چنین توابع بزرگی اجتناب ناپذیر است، بایستی گروه هی از توابع ایجاد شده و به صورت جداگانه مورد تجزیه و تحلیل قرار گیرد. با گراف های با ارتباطات متقابل بسیار بزرگ، طرح بندی دایره ای می تواند تجسم سازی سریع تری را فراهم سازد.



Circular layout view. Circular layouts are fast in comparison to hierarchical and orthogonal.

گرچه ممکن است گراف فوق چندان هم مفید نباشد ولی شکیل بوده و می تواند از نقطه گذاری و فیلترینگِ دستیِ توابع و بلاک های پرمرجع (frequently referenced) **ولی غیرمهم، مورد استفاده قرار گیرد. به عنوان آخرین مثال، به نمایش قائمِ خوشه ای** (cluster orthogonal) **تولید شده زیر توجه کنید:**



The cluster orthogonal view is used to visualize state maps and group function nodes together.

ابزارهایی که توابع مرکب چندگانه را نشان می دهند (از قبیل pg_graph_cat)، گره ها را با استفاده از توابع موجود در خود، به صورت خوشه هایی دسته بندی می کنند. این نمایش برای جداسازی توابع به صورت بصری بر روی صفحه کامپیوتر مفید هستند، در حالی که به محقق این امکان را می دهند که ارتباط بین آنها را بتواند ببیند. پنجره Explorer در GDE، درخت های برچسب دارِ قابل اضمحلال (collapsible labeled trees) را می سازد که امکان جستجو و دستیابی سریع به توابع و گره های مهم آنها را فراهم می سازد.

نویسنده: پدرام امینی (pedram.openrce.org)

مترجم (ترجمه ی آزاد): سعید بیکی (cephexin@secumania.net)

Secumania Security & Vulnerability Research Lab
www.secumania.net