ابزاری برای تشخیص و آنالیز نرم افزارهای مضر

چکیده

در این مقاله، ابزاری را به نام PEAT ارائه می دهیم: Portable Executable Analysis Toolkit؛ ابزاری جهت آنالیز فایل های PE.

PEAT نرم افزاری است حاوی از مجموعه ای از ابزاری که ممکن است یک آنالیزر برای بررسی جنبه های ساختاری یک فایل Windows Portable Executable (یا PE) آنها را استفاده کند تا به این وسیله تعیین کند آیا پس از کامپایل، کدهای مضر در application اضافه شده اند یا خیر. این ابزار مبتنی بر ویژگی های ساختاری از executable ها (فایل های اجرایی) هستند که ممکن است وجود کدهای مضر اضافه شده را نشان دهد. قضیه اصولی در رابطه با این ابزار این است که بسیاری از برنامه، از ابتدا تا انتهای کد، به صورت یک فایل باینری و همگن کامپایل می شوند و دارای ویژگی های ساختاری مشخصی می باشند. هر توزیع از این ساختار همگن، نشانه ای قوی خواهد بود در راستای دستکاری شدن فایل باینری. برای مثال، شاید این فایل بهمراه یک ویروس یا تروجان باشد. ما بررسی های خود را روی آنالیز ویژگی ساختاری متمرکز می کنیم. شما نیز می توانید مفاهیم و نظریه های موجود در PEAT را توسعه دهید.

مقدمه

نرم افزارهای مضر، تهدید عمده برای سیستم های اطلاعاتی امروزه شناخته می شود. تشخیص و آنالیز برنامه های خطرناک، اغلب کاری پر هزینه و پر نقص می باشد. دشواری این عمل بوسیله یک مسابقه challenge تاکید شد. در این مسابقه در یک باشگاه امنیتی، رفتارهای مضر یک برنامه بخصوص را مورد بررسی قرار می دادند (لازم به ذکر است که در مسابقه، مضر بودن عملیات این برنامه از قبل نیز معلوم بوده است).

اغلب تشخیص یک برنامه (یا بخشی مرتبط با آن) بعنوان یک عامل مضر و مخرب نیمی از این مبارزه است. در این مقاله، ما ابزار اولیه ای را معرفی خواهیم کرد که ما را در آنالیز نرم افزارهای بالقوه مضر کمک می کنند. در این سطح از کار، روی تشخیص نرم افزارهای مضر (malware) که به یک برنامه (application) میزبان و مفید دیگر ضمیمه شده اند تمرکز می کنیم. این روشی است که برای بسیاری از انواع malware ها مورد استفاده قرار می گیرد که شامل ویروس ها و بسیاری از برنامه های تروجان نیز می شود. در زمانی که برنامه های ویروس یا تروجان عملیات مخربی را خارج از آگاهی کاربر انجام م دهند، برنامه میزبان نقش نوعی پوشش را بازی خواهد کرد. این برنامه ها معمولا تکثیر می یابند و این کار هنگام پیوست به بازی ها یا دیگر فایل های اجرایی فریبنده یا پرکاربرد محسوس تر است.

برنامه نویس های مخرب، خلاقیت خود را در این زمینه با توسعه تعداد زیادی از تکنیک ها که یک malware می تواند به یک میزبانِ مفید ضمیمه شود نشان داده اند. چندین روش معمول برای Insertion یا الحاق وجود دارد که عبارتند از: اضافه کردن بخش های جدید به یک فایل اجرایی، اضافه کردن کدهای مضر به آخرین بخش از فایل میزبان یا یافتن یک ناحیه استفاده نشده از بایت ها درون host و سپس نوشتن محتویات مضر در آنجا می شود. یک روش ظریف ولی موثر از روش های الحاق این است که بخش هایی از برنامه میزبان را جاینویسی کنیم (overwrite کنیم).

حتی با در دست روش های بیشماری که در ضمیمه کردن یک برنامه مضر به یک برنامه میزبان وجود دارد، معمولا در این فرآیند، حتی تعیین نقطه آلودگی (نقطه ای که کدها را به آنجا اضافه کنیم) نیز کاری بسیار وقت گیر خواهد بود. با در درست داشتن این نقطه، ابزارهای سنتی از قبیل Disassembler ها و Debugger ها، برای بررسی malware ها مفید واقع می شوند، اما به یک آنالیزر، کمک خاصی نخواهند رساند.

تشخیص نرم افزارهای مضری که در یک قسمت داده ای یا دیگر قسمت های غیرمنتظره پنهان می شوند سخت تر خواهد بود. نکته دیگر اینکه، اندازه کلی کد برای یک برنامه مضر، اغلب کوچکتر از اندازه میزبانی هستند که آلوده می کنند. برای اینکه به یک آنالیزر نرم افزارهای مضر کمک کنیم که به سرعت و کاملا موثر malware را درون یک برنامه میزبان تعیین محل کنید، ابزاری با نام Portable Executable Analysis Toolkit یا PEAT را توسعه دادیم. هدف PEAT این است که فایل های Microsoft Windows Portable Exeutable (PE) را جهت نشانه هایی از کدهای مضر بازرسی کند. ما این عمل را با توسعه تکنیک های آنالیزی انجام می دهیم که بخش های یک برنامه را تشخیص می دهد و آنهایی که در مقایسه با برنامه میزبان اصلی تفاوت دارند را شناسایی می کنیم.

وجود چنین ناحیه هایی، نشانه ای مستحکم دال بر آلودگی برنامه میزبان توسط نرم افزار مضر می باشد. این مقاله به صورت زیر سازمان دهی شده است. ابتدا تکنولوژی های موجود را برای تشریح و شناخت کدهای غیرمطلوب (ویروسها، برنامه های backdoor و ...) بطور خلاصه بین می کنیم. سپس قابلیت هایی را که PEAT ارائه می دهد (بهمراه مفاهیم پنهان در این قابلیت ها و کاربردهای مورد نظر برای آنها) توصیف می کنیم. این مقاله مطالعات و تجربه های آزمایشی ما را در استفاده از ابزار PEAT جهت بررسی یک تروجان خطرناک با نام BackOrfice2000 نشان می دهد که در یک برنامه به ظاهر بی ضرر پنهان شده است.

بخش دیگر مقاله، نقاطه ضعف PEAT را بیان می کند. سپس کار را با یک بخش نتیجه گیری بپایان می رسانیم که اهداف ما را در توسعه PEAT جهت افزایش قابلیت آن در آنالیز نرم افزارهای مضر نشان می دهد.

پیش زمینه

در نگاه کلی، تشخیص نرم افزارهای مضر از لحاظ فرضیات علمی غیرممکن است. اما از نگاه، یعنی جستجوی یک نمونه کد مضر، این کار نه تنها غیر ممکن نیست، بلکه روزانه توسط نرم افزارهای AntiVirus نیز انجام می شود. لذا، ما راه حل های تجاری خوبی جهت تشخیص نمونه کدهای مضر و شناخته شده داریم (البته برای تشخیص کدهای ناشناخته باید تحلیل های بیشتری انجام شود). پس این خط مشیء ها و راه حل ها را می توان در توسعه نرم افزارهای ضدویروس و ... به کار برد. اما از نگاه کلی در تعیین اینکه آیا یک نرم افزار دارای عاملیت مضر است یک مشکل وجود دارد و آن غیرقابل تصمیم بودن آن است. منظور این است که ما نمی توانیم یک application معین را مورد بررسی قرار داده و به صورت کلی تصمیم بگیریم که آیا حاوی کدهایی هست که رفتار مخرب و مضر را نتیجه بدهد.

این مسئله معادل با مسئله غیرمداومی در تئوری علم کامپیوتر است که می گوید هیچ الگوریتم همه جانبه ای وجود ندارد که قادر به تعیین رفتار یک برنامه دلخواه باشد. گذشته از مسئله غیرمداوم، ما تعریفی از رفتار مضر و مخرب نیز در ذهن خود داریم. اعمال مضر روی مجموعه وسیعی از زمینه ها اعمال می شوند. برای مثال، یک برنامه که دیسک را فرمت می کند ممکن است دقیقا همان چیزی باشد که کاربر قصد انجام آن را دارد (و لذا نمی توان آنرا یک رفتار مخرب بحساب آورد)، اما هنگامی که مثلا این عمل در یک screensaver جاسازی شده و بدون اطلاع کاربر انجام شود، ما آنرا رفتار مخرب می دانیم. بنابراین، ما نمی توانیم الگوریتمی را جهت تصمیم راجع به مخرب بودن اعمال توسعه دهیم. دیگر کارهای ابتدایی که در

این زمینه انجام شده است، خصوصیات غیرقابل تصمیم بودن (یا تصمیم ناپذیری) نرم افزارهای مضر را در زمینه هـای مختلـف از دیدگاه کلی ثابت کرده اند.

اکنون با اینکه با توضیحات قبل ثابت شد که تشخیص کدهای مضر از دیدگاه کلی تصمیم ناپذیر است، چه گزینه هـایی در ایـن مسئله یعنی پیدا کردن کدهای مضر شناخته وجود دارد؟

یک راه جهت تشخیص کدهای مـضر در برنامـه هـای قابـل اجـرا (executable)، رسـیدگی بـه روش گـروه تحقیقـی LSFM می باشد. در این راه، هم روش های پویا و هم روش های ایستا جهت بررسی مدل[1] اعمال می شود تـا اطمینـان حاصـل کنیم برنامه تحت آنالیز هیچ گونه تجاوزی به سیاست های امنیتی تعیین شده ندارد.

این مثالی از یک خط مشیء در تشخیص نرم افزارهای مضر است که آنـالیزر رفتارهـای مخـرب را در آن بـه صـورت یک سیاست (policy) تعریف می کند. تحقیقات ما با خط مشیء های سنتی تفاوت دارد. در تحقیقات ما برای تشخیص کدهای مخرب نیازی به تعریف یا تشخیص *رفتارهای* مخرب نیست، بلکه روی مشخـصات *ساختاری* در کـدهای قابـل اجـرا و مخـرب تمرکز دارد. این خط مشیء امکان بررسی هر application (چه از قبل شناخته شده باشد یا خیر) را فراهم می سـازد. در ایـن خطی مشیء ما تعیین می کنیم که آیا application مورد نظر نسبت به توسعه اصلی خود مذاکراتی پنهانی با دنیای خارج خـود دارد یا خیر (که در این صورت درمی یابیم که با نسخه اصلی خود تفاوت هایی دارد).

چنین مذاکراتی معمولا به علت یک ویروس یا تروجان جاسازی شده بوده که در حـین اجـرای بعـدی از برنامـه قابـل اجرا فعال شده اند.

ما در این مقاله تنها پلاتفرم Microsoft Windows را بـرای PEAT مـورد بررسـی قـرار مـی دهـیم، چـرا کـه ایـن پلاتفرم تسلط خود را در بازار ثابت کرده و کدهای مخرب معمولا این سیستم عامل و application های آنرا هـدف قـرار مـی دهند. البته استفاده از پلاتفرم های دیگر نیز در حال افزایش است و در نتیجه حملات مخرب روی آنها نیز وجود دارند، لـذا مـا PEAT را طوری توسعه دادیم که بسادگی آنرا بتوان برای سازش با دیگر فرمت های اجرایی فایل از قبیل ELF نیز بکاربرد.

ابزار PEAT : ابزاری برای آنالیز فایل های PE

PEAT به یک آنالیزر ابزاری را ارائه می دهد تا بتوان فایل های PE را جهت نشانه هایی از کـدهای مخـرب بازرسـی کرد. فعالیت های بیشتر سبب توسعه PEAT شده و قابلیت هایی را ارائه خواهد کرد که بتوان ویژگـی هـا و امکانـات آن نـرم افزار را درک کرد. این ابزار برای تعیین محل ویژگی های ساختاری از فایـل هـای اجرایـی طراحـی شـده اسـت کـه در نـواحی اطراف قرار نمی گیرند، یعنی ناحیه هایی از بایت ها که در یک فایل باینری همگن اضافه شده اند.

حکم کلی در رابطه با این کارکرد این ابزار این است که برنامه ها معمولا به یک فایل باینری ثابت کامپایل مـی شـوند. هر گونه انحراف و دستکاری از این ثبات دلیلی محکم برای دستکاری آن می باشد. این امکـان وجـود دارد کـه برنامـه بـا یـک ویروس آلوده شود یا حاوی یک برنامه تروجان باشد یا اینکه طوری دستکاری شده باشد که رفتارهایی غیـر از اهـداف برنامـه نویس انجام شود.

[1] Model Checking

مرور

ابزارهای PEAT در سه گروه دسته بندی می شوند: بررسی هایی ایستا و ساده، تصویر سازی و آنالیز آماری و اتوماتیک.

بررسی های ایستا و ساده حاوی لیستی از قابلیت هایی است که حضور یا غیاب کوشش های PEAT برای حقیقت یابی به منظور بدست آوردن سریع اطلاعات را در اختیار می گذارد که ممکن است اشاره بر مظنون بودن کد داشته باشد.

برای مثال، اگر چنانچه نقطه ورودی برنامه، در یک موقعیت غیرمعمول باشد، ابزارPEAT بلافاصله اخطار می دهد.

ابزارهای تصویرسازی شامل ترسیم گرافیکی از چند ویژگی موجود در فایل PE می باشد که برای نمونه می توان به مثال هایی چون، احتمال شناسایی ناحیه های بایتی را در فایل PE که حاوی کد، داده های اسکی، padding و یا مقادیر بایتی تصادفی مربوط به offset های آدرس برای دستورهایی که اعمالی چون jump، call و یا ثبت الگوهای دستیابی دستورهایی که رفتار خاصی را نشان می دهند اشاره کرد (یعنی push کردن آرگومان ها بروی پشته و برقراری یک فراخوانی).

ابزار تصویر سازی از Disassembler موجود درPEAT نیز برای تفسیر (parse) و رمزبرداری (decode) دستورها استفاده می کند (این کار بر اساس کاری است که توسط Watanabe انجام شده است). آنگاه کاربر ممکن است لیست disassembly را ببیند. بعلاوه، به منظور تشخیص رشته های اسکی، کاربر ممکن است نمایش اسکی از تمام مقادیر بایتی درون یک ناحیه معین را بنمایش بگذارد. بعلاوه، این ابزار تصویرسازی به این منظور طراحی شده اند که به یک آنالیزر حرفه ای اجازه کاوش روی یک فایل اجرایی را می دهد. به این طریق شناسایی ناحیه هایی که با کل برنامه در تناقض هستند راحت تر می شود. برای تکمیل این قابلیت های آنالیز،PEAT ابزارهای آنالیز اتوماتیکی را همچنین ارائه می دهد که بدین وسیله آنالیزر می تواند نواحی مشکوکی از فایل PE را بررسی نماید. ابزار آنالیز اتوماتیک مربوط به PEAT، آزمایش های آماری را انجام می دهند تا بدین وسیله نواحی غیرعادی مشخص گردد. آنالیز روی بسیاری از ویژگی های یکسانی از فایل PE به صورت ابزار تصویرسازی عمل می کند. کاربر ویژگی هایی برای رسیدگی انتخاب می کند، سپس موتور آنالیز بخش های فایل PE را به چندین ناحیه تقسیم کرده و تعیین می کند که آیا تفاوت های آماری محسوسی بین این بخش ها وجود دارد یا خیر. هر مورد غیرمتعارف به صورت یک ناحیه مشکوک گزارش داده شده و ذخیره می گردد و توضیحاتی نیز بصورت اتوماتیک بهمراه آنها تولید خواهد شد (که مشکوک بودن آن ناحیه را توصیف می کند).

بررسی های ایستا

PEAT چندین بررسی ایستا از فایل PE را تحت آنالیز انجام می دهد، تا بدین منظور به اطلاعاتی که ممکن است اشاره بر مشکوک بودن کد (یا قسمتی از آن) داشته باشد دست یابد. اولین مورد، یک بررسی روی آدرس نقطه ورودی برنامه می باشد که از هدر مربوط به فایل PE بدست می آید. این آدرس بایستی در بخشی قرار بگیرد که بصورت اجرایی یا executable علامت گذاری شده باشد (که معمولا اولین بخش خواهد بود و نام آن text. یا CODE می باشد). به هر حال، اگر آدرس روی آن بخش قرار نداشته باشد، مثلا اگر نقطه ورودی در بخشی reloc. قرار گیرد (که نبایستی حاوی کدهای اجرایی باشند)، آنگاه هنگامی که فایل PE درون PEAT بارگذاری شود، اخطاری در پنجره اصلی آن ظاهر خواهد شد.

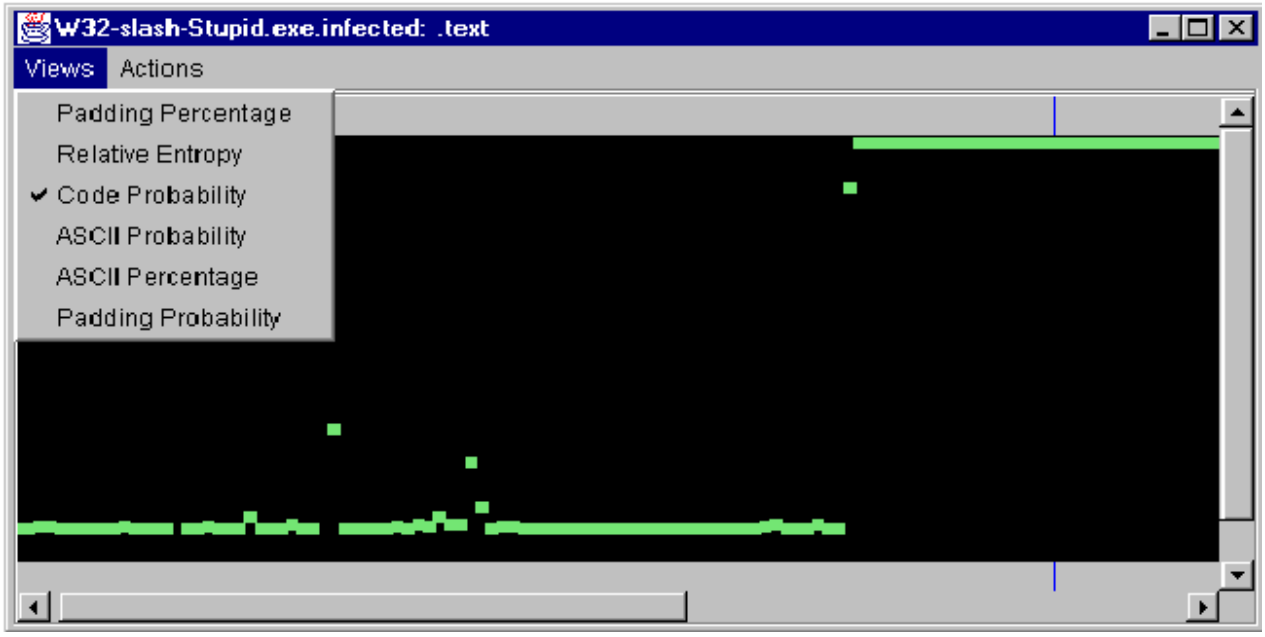
بررسی ایستای دیگری وجود دارد که فراخوانی های ساختگی[۲] را شناسایی میکند که فراخوانی هایی هستند که ما می خواهیم بلافاصله پس از چند دستور انجام شوند. یم چنین توالی از دستورها روش معمولی است که ویروس ها برای تعیین

---

[2] bogus call

آدرس خود در حافظه استفاده می کنند. به این دلیل که مقدار ثبات اشاره گر دستور[3] یا EIP بعنوان اثری از یک دستور CALL، روی پشته push می شود. ویروس این مورد را بلافاصله با pop کردن این مقدار اکسپلویت می کند.

بدلیل طبیعت مشکوک چندین توالی هایی از دستور، برنامه PEAT پس از disassemble شدن فایل PE به کاربر وجود این توالی ها را اخطار می دهد. سرانجام، PEAT تعیین می کند که چه کتابخانه های DLL ای در جدول ورودی[4] فایل PE لیست شده اند و نام و محتویات هریک را گزارش می دهد. بعلاوه، تمامی دستورهای موجود در برنامه را که تابعی را در یک کتابخانه DLL فراخوانی می کنند پیدا کرده و مکان آن دستورها را بهمراه نام کتابخانه و تابع گزارش می دهد.



تصویر ۱: نمایش بخش به صورت Byte-Type: احتمال کد

این یک گذر ابتدایی و سریع در تعیین قابلیت های غیرمنتظره در برنامه می باشد. از جمله این قابلیت ها ورودی یا خروجی فایل یا شبکه در یک application می باشد، در حالیکه شاید آن برنامه به این عاملیت نیاز نداشته باشد.

ابزار تصویرسازی

نمایش Byte-Type: ابزار تصویرسازی چندین راه برای نمایش ویژگی های ساختاری یک فایل PE ارائه می دهد. یکی از این نمایش ها، طرحی است که به آنالیزر اجازه می دهد تا بفهمد کدام نواحی از یک فایل PE حاوی کد، داده های ASCII، داده های Padding برای ترتیب دهی یا مقادیر بایت تصادفی هستند. یک مثال در تصویر ۱ نشان داده شده که قطعه text. از ویروس W32.Stupid در آن نمایش داده شده است. هر نقطه در امتداد محور افق، پنجره ای از بایت ها از قطعه text. را نشان می دهد و مقدار آن در امتداد قائم از ۰ تا ۱ مقیاس بندی شده است که نشان می دهد پنجره بایت ها با چه احتمالی شامل یک مورد از نوع بایت هست که در این صورت مقادیر بزرگتر احتمال بیشتری را نشان می دهند.

---

[3] Instruction Pointer Register
[4] Import Table

در این مثال، احتمال کد نمایش داده شده است. چه چیزی که می بینیم این است که فقط آخرین بخش از این قطعه .text شامل کد واقعی است که با استفاده از خط پررنگی از نقاط که نسبت به محور قائم در وضعیت بالایی قرار دارند نشان داده شده است.

مقادیر احتمالی توسط آزمایش های آماری استاندارد و متناسب[5] تعیین گشته است که در آنها مجموعه ای مشخص از مقادیر بایتی (مثلا، مقادیر موجود در محدوده کاراکتر اسکی) که در یک پنجره قابل رویت هستند محاسبه می گردند.

```
.text   0x4027C0 - 0x402E00

  0: ............T.h.i.s. .i.s..............*.S. .T. .U. .P. .I. .D.*.......V.i.n.3.2...S.t.u.p.
 37: i.d.....4...C.a.n.e.r.o.n. .D.i.a.z. .i.s. .a. .g.o.d.d.e.s.s.!.........V.i.c. .s.a.y.s.....
 75: ........J.E.R.K.!.......A.u.t.o.r.u.n...1.n.f.......[.a.u.t.o.r.u.n.].......O.P.E.N.=.S.M.I.
119: L.E...E.X.E.....VBA5.DLL....__vbaFPInt..__vbaVarTstGt...__vbaVarCmpGe...__vbaAryDestruct....
171: __vbaLenVar.__vbaFpI4...__vbaVarSub.__vbaVarMul.__vbaVarInt.__vbaStrVarCopy.__vbaAryConstruc
232: t...__vbaExitProc...__vbaVarDup.__vbaFileClose..__vbaPrintFile..__vbaFileOpen...__vbaInStrVa
278: r...__vbaVarAdd.__vbaInStr..__vbaStrVarMove.__vbaNew2...__vbaEnd....__vbaVarOr..__vbaI2Var..
333: __vbaVarCmpEq...__vbaVarAnd.__vbaBoolVarNull....__vbaVarTstNe...__vbaStrVarVal..__vbaFreeVar
394: List....__vbaVarTstEq...__vbaVarForNext.__vbaStrCat.__vbaStrCopy....__vbaStrCmp.__vbaLenBstr
453: ....__vbaVarForInit.__vbaFreeStrList....__vbaStrToUnicode...__vbaFreeStr........__vbaSetSyst
500: emError.__vbaStrToAnsi..__vbaI4Var..__vbaVarCopy....__vbaVarCat.__vbaFreeObj....."@.(c@.__vb
558: aVarMove....__vbaStrMove....__vbaFreeVar....__vbaHresultCheckObj....__vbaOnError....V..7. ..
612: ._DEST..U..7. ..._DEST..\...........4...+00...#C:\.........................L...`...p.......
659: .......................[..7. ..._DEST..Z..7. ..._DEST..\...........4......................
702: ................L...`...p.............................__vbaVarCmpNe.........@.............
744: ..............@....................`.-@.............../.................`(-@.............../..
788: ...............`0-@.............00................`$-@.............../.................`,-@.
832: ............./.................`.
```

تصویر ۲: نمایش ASCII

.text 0x401384 - 0x401CE4

Address	Bytes	Instruction	Operand
004015A0	8B 44 24 04	mov	eax,[esp]
004015A4	50	push	eax
004015A5	E8 C9 B7 00 00	call	040CD73
004015AA	59	pop	ecx
004015AB	C2 04 00	retn	0004
004015AE	90	nop	
004015AF	90	nop	
004015B0	55	push	ebp
004015B1	8B EC	mov	ebp,esp
004015B3	6A FF	push	FF
004015B5	68 E8 35 41 00	push	004135E8
004015BA	64 A1 00 00 00 00	mov	eax,[00000000h]
004015C0	50	push	eax
004015C1	64 89 25 00 00 00 00	mov	dword ptr ds:[00000000h],esp
004015C8	83 EC 14	sub	esp,14
004015CB	53	push	ebx
004015CC	56	push	esi
004015CD	57	push	edi
004015CE	8B F1	mov	esi,ecx
004015D0	33 FF	xor	edi,edi
004015D2	89 65 F0	mov	[ebp-10h],esp

تصویر ۳: نمایش Disassembly

[5] standard statistical proportion tests

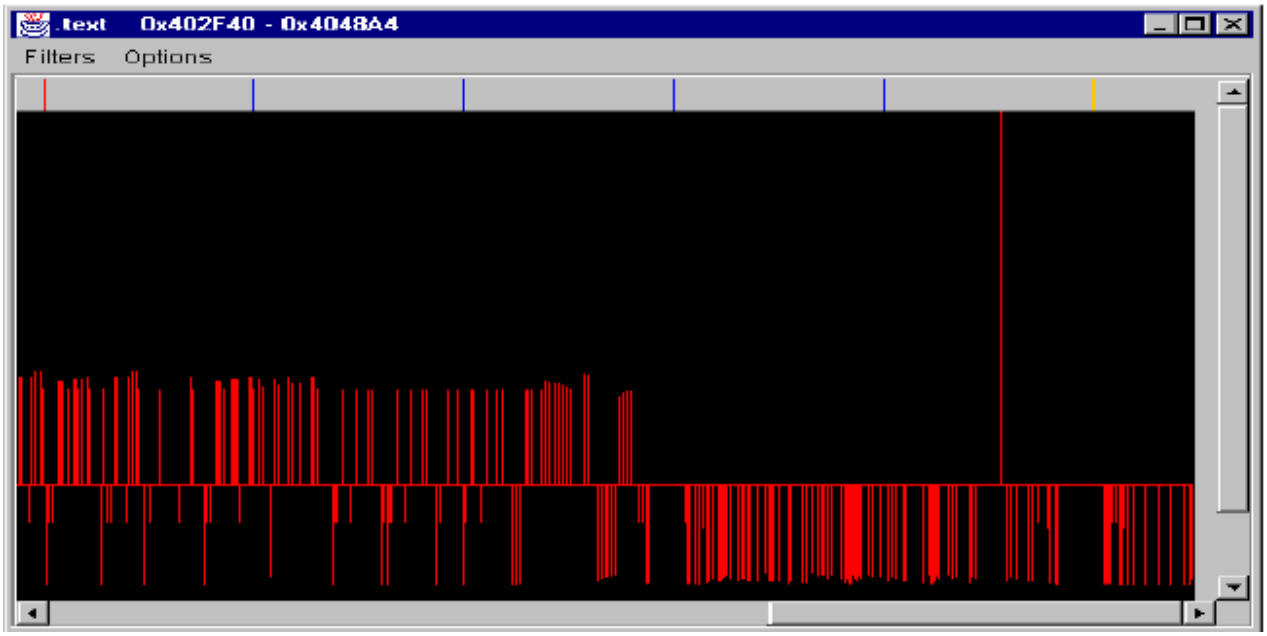
دیگر انواع اطلاعات که در این نمایش قابل دسترس هستند، طرحی از آنتروپی مقدار-بایت می باشد. این آنتروپی، بخشی است که به چندین پنجره تقسیم شده و آنتروپی کل مقدار بایتی در آن پنجره محاسبه می گردد. سپس این مقادیر آنتروپی دربرابر کل مقادیر آنتروپی برای هر پنجره نرمالیته شده و سپس روی محور قائم طرح ریزی می شوند.

نمایش ASCII (اسکی): از نمایش سطح-بخشی که در بالا توصیف شده است، کاربر ممکن است ناحیه ای را برای بازرسی های بیشتر انتخاب کند. یم نمایش اضافی در این راستا، جلوه های نمایشی اسکی برای هر بایت در ناحیه انتخاب شده می باشد. یک مثالی در تصویر ۲ نشان داده شده است. این بایت ها متناظر با ناحیه موجود در تصویر ۲ هستند که احتمال زیادی در دارا بودن داده های اسکی دارند.

نمایش Disassembly: راهی دیگر برای بازرسی یک ناحیه خاص این است که PEAT آن ناحیه را disassemble کرده و نتایج را نشان دهد. برای هر دستوری که تفسیر می گردد، ادرس، مقادیر بایتی خام، نام دستور و عملگرهایدستور نمایش می یابند. یک مثال در تصویر ۳ نشان داده شده است.

دستیابی به حافظه از طریق Offset های ثبات: ابزارPEAT نمایشی را ارائه می دهد که به کاربر اجازه می دهد تا بفهمد چه موقع حافظه با اضافه کردن یک offset به یک مقدار ثبات قابل دستیابی است تا به این صورت یک آدرس را در حافظه تعیین کند. نخست کاربر یک ثبات را برای رسیدگی انتخاب می کند - مثل ثبات EBP [6] - و سپس PEAT از اطلاعات disassembly برای یافتن و طرح ریزی کردن تمام آدرس های حافظه ای به این نوع، استفاده می کند. یک مثال در زیر نشان داده شده است. یک خط افقی از میان محور عمودی عبور کرده است و مقدار ۰ را نمایش می دهد (طول محور y ها برابر با صفر می باشد).

این نمایش را می توان مورد استفاده قرار داد تا به صورت تصویری تشخیص داد که کدام نواحی، متفاوت از نواحی دیگر (یعنی offset های بزرگتر، offset های تکراری بیشتر یا offset ها در جهات مخالف) از این وسیله برای دستیابی به حافظه استفاده می کنند.



تصویر ۴: نمایش Offset های ثبات

[6] Base Pointer Register

دیگر نمایش ها: ما نمایش های دیگری تعریف کرده ایم که هنوز درون PEAT اعمال نشده اند. یک چنین نمایشی offset های دستورات jump و call را نشان می دهد (مشابه نمایش offset های ثبات) تا تعیین کنیم کدام نواحی خود محتوا هستند و آیا jump های طولانی انجام شده است یا خیر (از قبیل بیرون از مرزهای قطعه اجرا شونده).
نمایش دیگری وجود دارد که الگوی های معمول دستوری را نشان می دهد، مثل وجود چند دستور push بعد از call (که push کردن آرگومان ها به روی stack به منظور انجام یک فراخوانی رویه ای را نشان می دهد). آنالیزر بایستی بصورت چشمی وجود یا فقدان چنین الگوهای معمول را به خاطر بسپارد و تعیین کند که آیا یک ناحیه از بخش با دیگر ناحیه ها فرق دارد یا خیر.

آنالیز آماری

به عنوان یک مکمل برای ابزار بررسی های ایستا و ساده و تصویرسازی، PEAT قابلیت های آنالیزی را مبتنی بر استفاده از روش های آماری برای تشخیص نواحی غیرعادی درون یک فایل PE ارائه می دهد. کاربر ممکن است محدوده وسیعی از ویژگی ها را برای استخراج از برنامه انتخاب کند، مثل:

- تکرارهای دستوری (Instruction Frequency)
- الگوهای دستوری (Instruction Pattern)
- Offset های ثبات
- Offset های Call و Jump
- آنتروپی (بی نظمی) مربوط به مقادیر opcode
- احتمالات Code و ASCII

موارد بالا به تفصیل در ادامه توضیح داده شده اند و سودمندی آنها در تشخیص نواحی غیرعادی در یک برنامه نشان داده شده است. اما ابتدا ما خط مشیء آماری و کلی را مورد بحث قرار می دهیم که برای هر ویژگی مورد استفاده بعنوان داده ورودی اعمال می شود. هنگامی که PEAT آنالیز اتوماتیک خود را انجام می دهد، این کار را روی هر بخش از فایل PE تکرار می کند. بخش به صورت دستورهایی disassemble شده و سپس به η پنجره متوالی و گسسته با تعداد ثابتی از دستورها تقسیم می شود. معیارهای مهمی برای هر پنجره محاسبه می شود (مثلا آنتروپی مربوط به مقادیر opcode) که لیستی از مقادیر را نتیجه می دهد

$$X = (x_1, x_2, ..., x_n)$$

از این لیست، لیست دیگری از تفاوت ها

$$Y = (y_1, y_2, ..., y_{n-1})$$

محاسبه می شود که:

$$y_i = x_{i+1} - x_i$$

سپس PEAT این کار را روی پنجره ها تکرار کرده و برای هر پنجره تعیین می کند که کدام داده های متناظر بـه X اشاره می کنند که X یک قسمت آماری با داده های باقیمانده و اشاره گر به X می باشد. برای پنجره i، انحراف استاندارد $X\backslash$ x_i محاسبه می شود و تعیین می گردد که آیا x_i با دو تقسیم استاندارد پیدا می شود یا خیر!

وقتی که اینگونه نبود، پنجره بصورت یک مورد غیرعادی (غیرمتعارف) بهمراه یک احتمال آماری گـزارش مـی شـود که حداقل منعکس کننده احتمال تحقق پذیری نوعی انحراف از x_i است که از توزیع تجربی باقیمانده ها بدست مـی آیـد. ایـن رویه لیستی از پنجره ها را تولید می کند که می توان گفت در رابطه با پنجره های دیگر ایـن بخـش دارای آنتروپـی غیرعـادی است. در رابطه با نقاط داده ای Y، نیز رویه ای مشابه در پنجره مورد استفاده قرار می گیرد و لیـستی از پنجـره هـایی را تولیـد می کند که تغییرات محسوس و متوالی را در معیارهای مورد نظر نشان می دهد. برای مثال، اگر الگوهـای دسـتوری معمـول در اشاره به بخش مشاهده شوند و سپس همه چیزها به طور اتفاقی ناپدید شود، این پدیـده گـزارش مـی شـود. منطـق موجـود در ورای استفاده از نقاط X و Y این است که ممکن است نقطه X برای یافتن یک ناحیه غیرعادی در یک بخـش کـافی نباشـد کـه نیمه اول این بخش طبیعی و نیمه دیگر آن با کدهای مضر جاینویسی شده است. با فرض اینکه چـارچوب عمـومی بـرای آنـالیز آماری بکار رود، PEAT معیارهای متفاوتی را در اختیار ما مـی گـذارد تـا بتـوانیم مجموعـه از معیارهـا بـرای تـشخیص مـوارد غیرعادی بسازیم.

تکرارهای دستوری

نظریه ای که در بازرسی توالی های دستوری از یک پنجره به پنجـره دیگـر وجـود دارد، از یکـی از مقـدمات پایـه ای سرچشمه می گیرد و آن این است که ویروس ها اکثرا به زبان اسـمبلی نوشـته مـی شـوند، در صـورتی کـه application هـای میزبان معمولا از زبان های سطح بالا کامپایل می گردند. ما مطالعاتی روی این مورد انجام دادیم تا هـر دسـتوری کـه متناوبـا در برنامه های زبان اسمبلی بکار می روند ولی در کدهای کامپایل شده وجود ندارد را شناسایی کنیم. نتایج این مطالعـات لیـستی از دستورهایی بود که بصورت متناوب برای یافتن پنجره های غیرعادی محاسبه شده بودند. تصور کنید کـه ناشـی از یـک فقـدان ناخواسته از دستورهای متناوب کامپایل شده، کد اسمبل شده و مضری که در یک بخش از فایل PE تزریق شـده بـود در خـلال آنالیزهای آماری کشف شود، در این هنگام آنالیزهای بیشتر نشان می دهند که دستورهای زبان اسمبلی بطور غیرعـادی در آن ناحیه مورد استفاده قرار گرفته اند.

الگوهای دستوری

انگیزه ای که برای بررسی الگوهای دستوری وجود دارد، بسیار شبیه بـه نظریـه هـای موجـود در بررسـی تـوالی هـای دستوری است. فرض ما این است که کد کامپایل شده احتمال به نمایش گذاشتن تـوالی هـای مـنظم دسـتوری را دارد تـا از آن طریق ساخت های معمولی چون فراخوانی های توابعی، برگـشت هـا و سـاختارهای حلقـه ای را انجـام دهـد. معاهـدات برنامـه نویسان زبان اسمبلی برای انجام این موارد ضرورتا مشابه کامپایلر نیست و احتمالا از یک اسـتفاده بـه اسـتفاده دیگـر پایـداری خود را از دست می دهند.

ما مطالعات مقدماتی راجع خروجی زبان اسمبلی از کامپایلر MS VC++ انجـام دادیـم و لیـستی از الگوهـایی کـه از اسـتفاده از ساختارهای معمول در زبان سطح-بالا نتیجه می شود را تهیه کردیم. توالی الگوها معیاری است که کاربر مـی توانـد آنـرا بـرای آمیختن در یک آنالیز انتخاب کنید که این کار با هدف تشخیص کد تزریق شده و مـضر در زبـان اسـمبلی انجـام مـی شـود (بـا استفاده از فقدان تصادفی برای چنین الگوهایی).

دستیابی به حافظه از طریق Offset های ثبات

ما قضیه دیگری را اثبات کردیم و آن اینکه application ها و کدهای مضر معمول، هر کدام ثبات های مختلف و مشخصی را مورد استفاده قرار می دهند. بالاخص اینکه ثباتEBP معمولا توسط برنامه های معمول بعنوان یک نقطه ارجاعی جهت دستیابی به متغیرهای محلی روی پشته استفاده می شود. برنامه های مضر، می توانند از این نقطه ارجاعی کلیدی برای تعیین محل آنها در حافظه استفاده کنند. بنابراین مقادیر offset ثبات که هنگام دستیابی به حافظه با استفاده از یک ثبات مورد استفاده قرار می گیرد، معیار دیگری است که می تواند در خلال آنالیزهای آماری مورد استفاده قرار گیرد.

فواصل Jump و Call

طرح بندی معمول برای یک application کامپایل شده از یک زبان سطح بالا، یک توالی از توابع خود محتوا می باشد. با استفاده از دستورات CALL و RET روندهای کنترل بین این توابع انجام می گردد. دستورات Jump، روند اجرا را درون یک تابع واحد تغییر می دهند و ساختارهای شرطی سطح بالا مثل جملات if و حلقه های loop را پیاده سازی می کنند. بنابراین، فواصل موجود در بین دستورهای jump برنامه بایستی نسبتا کوچک و منظم باشد و بهمین نحو فواصل بین دستورهای call بایستی نسبتا بزرگ و منظم باشد. چیزی که بایستی در برنامه های معمول کم دیده شود، فواصل شدیدا طولانی بین jump یا call می باشد،از قبیل دیگر بخش های فایل PE.

احتمالات Byte-Type

آخرین انواعی که PEAT بعنوان ورودی برای آنالیز آماری استفاده می کند، احتمالاتی است که پنجره ها شامل داده های اسکی، padding یا کد واقعی می باشند. این مشابه اطلاعاتی است که در نمایش Section ارائه می گردد. ارائه شد. در رابطه با دیگر معیارها باید گفت که این اطلاعات نوع-بایت می توانند در بازرسی های بیشتر روی نواحی نشانه گذاری شده بعنوان غیرعادی، استفاده شوند.

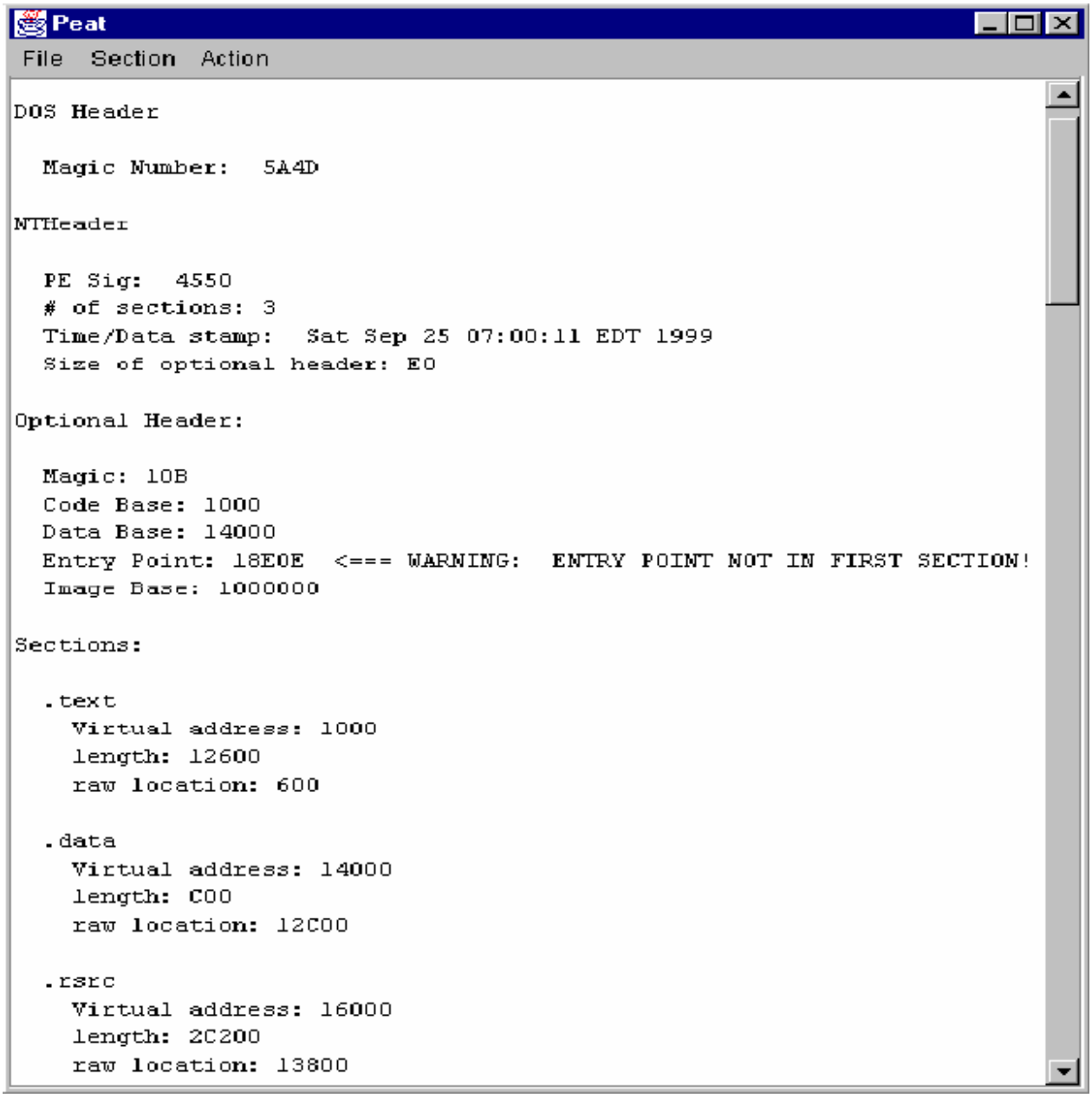
برای مثال، اگر یک پنجره بصورت outlier نشان داده شود (به این معنی که دارای فقدان ناگهانی الگوهای دستوری معمول است)، اما همزمان با آن بصورت یک outlier برای یک احتمال بالای تصادفی مبنی بر padding بودن و احتمال پائین مبنی بر code بودن نشان دهد، در این صورت آنالیزر با اطمینان کامل می تواند نتیجه بگیرد که عدم وجود الگوها نشانگر وجود کدهای زبان اسمبلی نیست ولی در عوض به معنی عدم وجود هر دو نوع کد است.

هنگامی که کل آنالیز اتوماتیک بپایان می رسد، به آنالیزر لیستی از پنجره هایی که مشکوک به غیرعادی بودن هستند ارائه می شود. هر پنجره با مکان خود در بخش و توضیحی از ویژگی هایی که مبنی بر این گمان هستند گزارش می شود. از این لیست، آنالیزر می تواند بسادگی گزینه های تصویر سازی مثلdisassembly را درگیر کار سازد تا بدین صورت بازرسی های بیشتری روی یک ناحیه بخصوص انجام دهد.

نتایج

ما موفقیتی ابتدایی در استفاده از PEAT جهت انجام آنالیز روی چندین نمونه کد مضر داشتیم. در یک مورد خاص نیز سرور Back Orficie 2000 را یافتیم که درون یک برنامه بظاهر بی ضرر پنهان شده بود. InPEct ابزاری است که برای تزریق تروجان های دلخواه به درون برنامه های دلخواه روی پلاتفرم ویندوز مورد استفاده قرار می گیرد. هنگامی که یک کاربر فایل اجرایی حاصل شده را اجرا می کند، تروجان تزریق شده در پشت صحنه شروع به فعالیت کرده و برنامه قربانی نیز به صورت طبیعی کار خود را انجام می دهد. هنگامی که برنامه قربانی پایان می پذیرد، برنامه تروجان به کار خود ادامه خواهد داد. ما از InPEct جهت تزریق سرور BO2K به درون یک برنامه دلخواه ویندوز استفاده کردیم. این برنامه ماشین حساب یا Calculator ویندوز می باشد که با نام calc.exe در محیط ویندوز قابل دستیابی است. سپس فایل اجرایی مورد تزریق را با PEAT بررسی کردیم تا تعیین شود آیا معیارهای موجود در PEAT قادر به تشخیص وجود تروجان هستند و می توانند بینشی از روش تزریق را به ما بدهند. فرآیند بررسی انجام شده بهمراه نتایج حاصله در زیر آورده شده است. هنگامی که PEAT ابتدا یک فایل PE را بارگذاری می کند، چندین قطعه اطلاعاتی از هدر PE را نشان می دهد که شامل لیستی از بخش های (section) فایل و نقطه ورودی (entry point) می باشد.

بمحض بارگذاری برنامه آلوده شده ماشین حساب، PEAT اخطاری را مبنی بر قرار داشتن نقطه ورودی برنامه در یک مکان غیرمعمول اعلان می دارد. کنترل برنامه در بخش rsrc. شروع می شود، در حالیکه انتظار معمول بخش text. است. بعلاوه، در مقایسه با برنامه های نمونه، مشاهده می کنیم که بخش rsrc. بطور غیرمعمولی طولانی است. این اطلاعات که در تصویر ۵ نشان داده شده اند، اولین مدرک جهت دستکاری شدن برنامه می باشد. در گام بعد PEAT، آنالیز اتوماتیک خود را انجام می دهد تا نواحی غیرعادی درون هر یک از بخش های فایل را تشخیص دهد. در این هنگام PEAT، دو پنجره غیرعادی را در بخش rsrc. تشخیص داده و آنها را همان طور که در تصویر ۶ آشکار است به نمایش می گذارد. اولین پنجره با معیارهای احتمال وجود کد سنجش می شود. احتمال زیادی دارد که این پنجره حاوی کد باشد، اما دیگر پنجره ها در این بخش حاوی کد نیستند. یک آنالیزر بایستی این پنجره را به چشم بدگمانی بنگرد، چرا که بخش rsrc. اصولا حاوی کد نمی باشد. به هر حال، این مورد با اخطاری که توسط PEAT مبنی بر غیرعادی بودن نقطه ورودی داده شد سازگار می باشد. احتمالا نقطه ورودی در این پنجره غیرعادی تمام می شود.

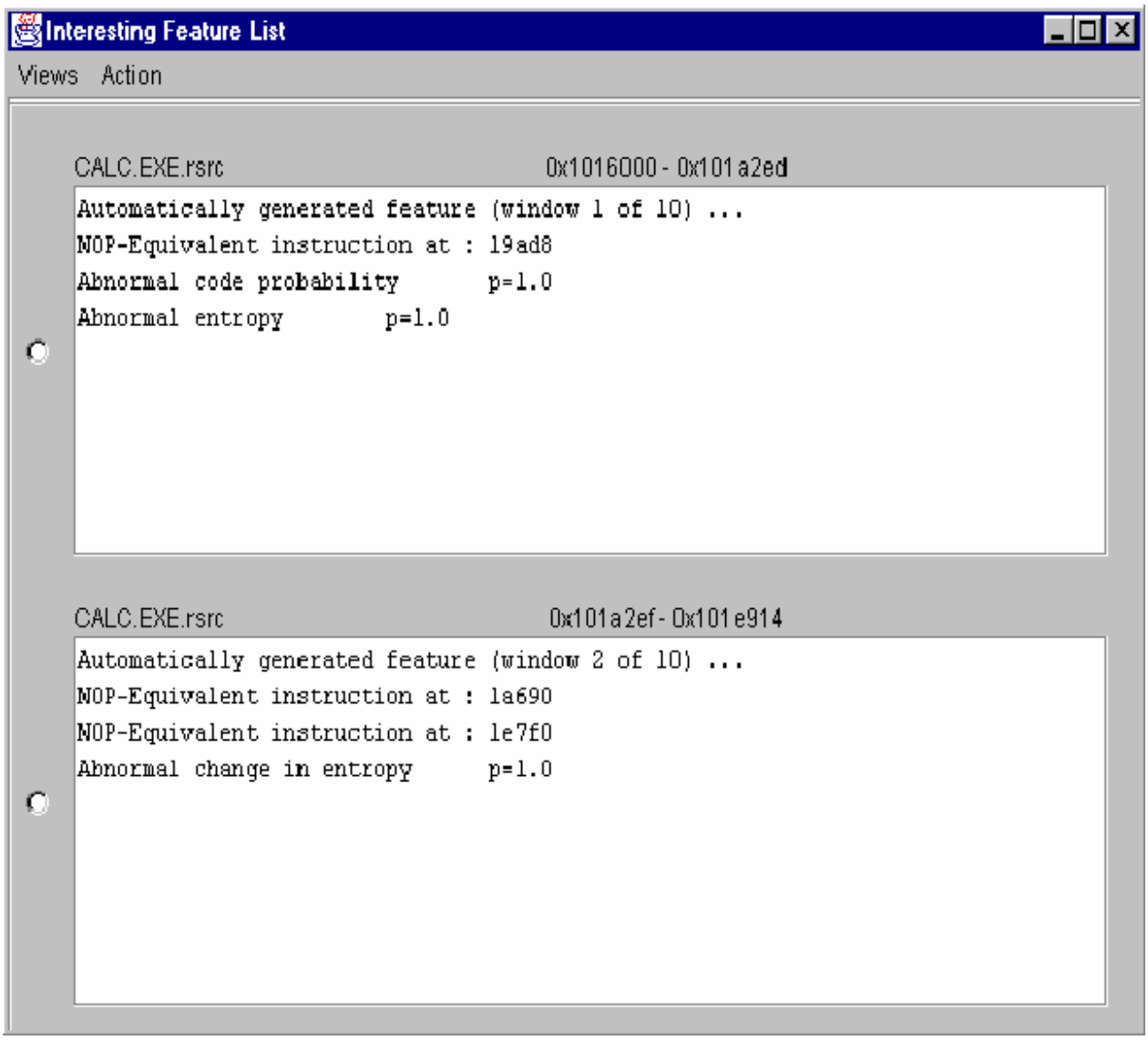


تصویر ۵: اطلاعات هدر PE برای برنامه آلوده شده ماشین حساب

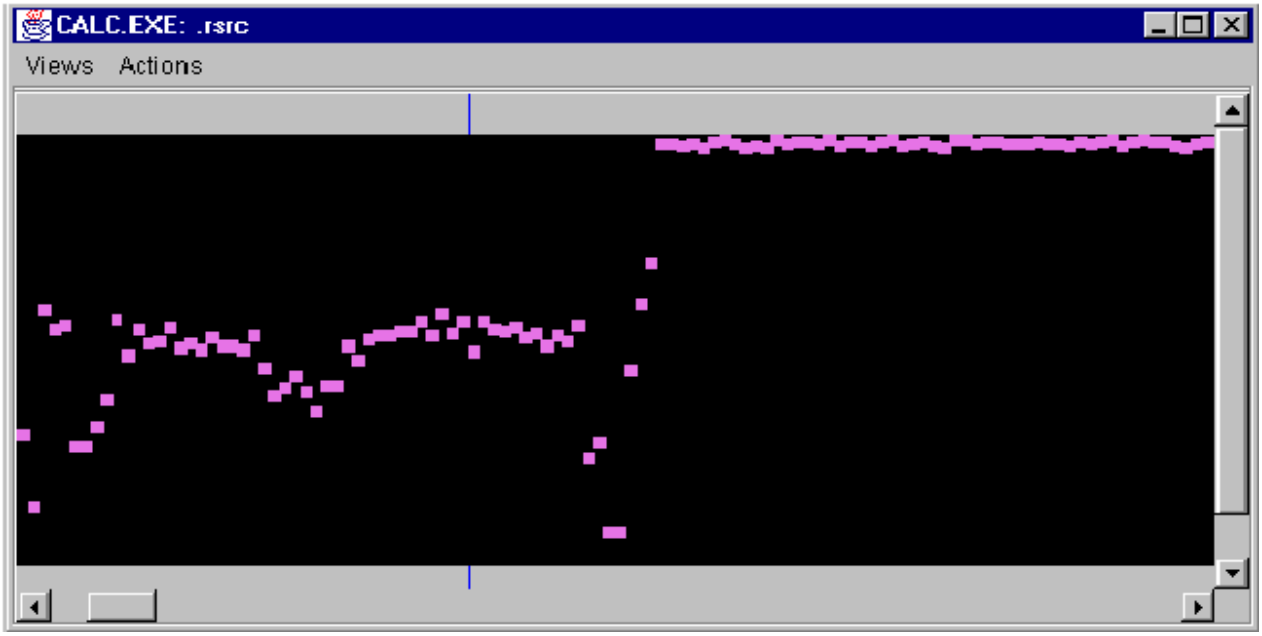
این پنجره به دلیل سطح غیرعادی آنتروپی از باقیمانده بخش متمایز است. دومین پنجره غیرعادی جهت داشتن یک تغییر غیرعادی در آنتروپی (در مقایسه با پنجره قبلی) علامت گذاری شده است. این دو اخطار (یک مقدار غیرعادی به همراه یک تغییر محسوس در آن مقدار) اشاره می کنند که چیزی درون برنامه اصلی اضافه شده است. برای بررسی بیشتر این سطوح آنتروپی، آنالیزر از ابزار تصویرسازی جهت بررسی بخش rsrc. استفاده می کند.

نمایش معیار آنتروپی برای بخش rsrc. در تصویر ۷ نشان داده شده است. مشاهده می کنیم که نقطه ای وجود دارد که سطح آنتروپی در آنجا، بصورت ناگهانی افزایش می یابد. این دلیلی است محکم بر این که یک ناحیه رمزنگاری شده از بایت در این نقطه وجود دارد. در این هنگام، آنالیزر مدارک کافی برای تشخیص یک الگوی بخصوص را بدست آورده است. تکنیک معمولی که ابزار تزریق تروجان از آن استفاده می کنند (که InPEct هم شامل آنها می شود)، رمزنگاری و اضافه کردن تروجان به انتهای آخرین بخش از برنامه مورد نظر می باشد. روش تزریق، یک کد دلخواه نیز به تروجان اضافه می کند تا

بتوان بعضی کارها را انجام داد، مثلا رمزگشایی تروجان، اجرای آن و اعطای کنترل به برنامه اصلی و عاقبت تغییر نقطه ورودی برنامه به این روتین الحاق شده Startup. پس از مشاهده تمام این مدارک، آنالیزر می تواند با اطمینان استنتاج کند که فایل تحت بررسی، با یک تروجان بوسیله روش توصیف شده در بالا، آلوده شده است.



تصویر ۶: دو ناحیه غیرعادی در بخش rsrc.



تصویر ۷: معیار آنتروپی برای بخش **.rsrc** از برنامه ماشین حساب

اگر آنالیزر علاقه مند به بررسی بیشتر روی فایل باشد، نمایش Disassembly از روتین الحاق شده Startup، به عنوان نقطه شروع مناسبی تجلی خواهد کرد. دیگر قابلیت ها برای یک آنالیزر در این سناریو مفید خواهند بود، از قبیل تعیین هویت یا قابلیت های تروجان الحاق شده. پیاده سازی فعلی از PEAT هنوز این مسائل را در برندارد، اما در بخش بعدی (بخش نقص ها) مورد بحث قرار گرفته اند.

نقایص

فناوری های تشخیص نرم افزارهای مضر از یک مشکل عمومی رنج می برند: هنگامی که یک نفوذگر ضابطه و معیاری که منطق تشخیص را کنترل می کند می فهمد، آنگاه می تواند حمله خود را جهت سازش با این معیار و در نتیجه شکست تشخیص انجام دهد. PEAT در بعضی مواقع نسبت به این مورد ضعف دارد. اگرچه PEAT مجموعه ای از چندین معیار مستقل را دارد، اما یک نفوذگر مصمم می تواند حمله ای را طراحی کند که از روش های تشخیص PEAT سرباز بزند. برای مثال، یک آنالیزر از PEAT برای تشخیص حمله هایی استفاده می کند و تنها از جنبه های مورد نظر از PEAT نسبت به فایل های اجرایی آلوده شده استفاده می کند. اگر فایل اجرایی میزبان کاملا با کد مضر جاینویسی و تعویض شود، PEAT هیچ تناقضی را در خلال آنالیز تشخیص نمی دهد. به هر حال، این حمله برای یک نفوذگر جالب نخواهد بود، چرا که کاربر قربانی بلافاصله بعد از اجرایی فایل میزبان مشکوک می شود (چون خصوصیت فایل میزبان و در نتیجه اعمالی که انجام می دهد تغییر یافته اند). اگر نفوذگر اطلاعاتی از کد منبع و تاریخچه تکامل یک برنامه میزبان در اختیار داشته باشد، می تواند کد مضر خود را به سبک مشابهی در آورد، بطوریکه معیارهای PEAT مقادیر موجود در کد مضر و برنامه میزبان را بسان هم می نگرد. برای مثال، نفوذگر می تواند تروجانی را در VC++ توسعه دهد که به صورت یک تابع وابسته به برنامه میزبان تغییر جلوه دهد (بطوریکه قابلیت توسعه آن تابع در آن زبان برای عموم شناخته شده باشد).

خوشبختانه این اولین قدم در جلوگیری از تشخیص در این وضعیت می باشد. دیگر فاکتورها از قبیل سطوح بهینه سازی کامپایلر و حتی سبک های کدنویسی ممکن است الگوی های برجسته تروجان را نمایش دهند. همچنین نفوذگر هنوز مشکل تغییر برنامه میزبان را دارد.

امکان آلوده کرده یک برنامه میزبان با یک قطعه کد بسیار کوچک وجود دارد. این قطعه کد، یک DLL مجزا حاوی payload های مضر را بارگذاری می کند یا شاید پروسه دیگری را ایجاد می نماید. PEAT در مقابله با این حمله ناتوان است. PEAT می تواند وجود فراخوانی ها به توابع DLL معمول از قبیل LoadLibrary() یا CreateProcess() را گزارش دهد. اما نمی تواند این روند را برای تمام کتابخانه ها یا فایل های قابل اجرایی مجرا انجام دهد (بطوریکه آنها را در زمینه فایل اصلی تحت آنالیز، بررسی و آنالیز کند). بعلاوه، حمله های دیگر روی فایل های اجرایی وجود دارند که خارج از اهداف PEAT می باشند. برای مثال، بعضی ویروس ها، image های اجرایی موجود در حافظه را مورد حمله قرار می دهند، در حالیکه PEAT تنها قابلیت کار با فایل های ذخیره شده روی دیسک را دارد. سرانجام اینکه، PEAT شاید موارد غیرمتعارفی را گزارش دهد که لزوما نشانه وجود کدهای مضر نباشند. برای مثال، شاید این گزارش را بدهد که در انتهای قطعه text.، آنتروپی مقدار بایت ناگهان تغییر می کند. بررسی های بیشتر شاید نشان دهد که این مورد ناشی از وجود پرکننده های (padding) تنظیم بخش[7] باشد و نه تغییر در فایل اصلی. PEAT را نمی توان به عنوان یک ابزار کاملا اتوماتیک بکار برد، چرا که آنالیزر به اندازه ای اطلاعات از فایل های PE، ویروس ها و دیگر مفاهیم در سطح سیستم (system-level) نیازمند است و برای انجام یک آنالیز کامل و بی نقص باید تجربه کارکرد با PEAT را نیز داشته باشد، چرا که باید چگونگی تفسیر خروج ها از معیارهای مختلف توسط این ابزار را بداند.

کارهای آتی

اجرای اصلی از پروژه تشخیص جنبه های ساختاری مختلف از فایل های اجرایی ویندوز بود که وجود کدهای مضر را نشان دهند. قابلیت بزرگی که ما در PEAT قرار دادیم، جزئی است که قادر به آنالیز ویژگی های یک ناحیه از کد است. PEAT قبلا تشخیص DLL های وارد (import) شده و مکان فراخوانی ها به توابع DLL را دارا بود. ما در آینده از این اطلاعات استفاده خواهیم کرد و تعیین خواهیم کرد که کد قادر به انجام چه اعمالی است. بعلاوه، قابلیتی را برقرار خواهیم ساخت تا بطور بازگشتی به DLL های وارد شده ناشناخته برسیم و ویژگی های آن را نیز تعیین کنیم. برای مثال، اگر فایل PE، یک DLL با نام unknown.dll را وارد می کند و تابع foo() در آن فراخوانی شده است، ما مسلما مایل هستیم که ویژگی های آن تابع را نیز بررسی کنیم! در کنار آنالیز ویژگی های کد، خاطر نشان شدیم که بسیاری از گونه های کدهای مضر، به خصوص ویروسها، از بخش های کد مربوط به دیگر برنامه های مضر استفاده می کنند. این بخش ها از کد که مجددا استفاده می شوند، نفوذگر را از نوشتن مجدد عاملیت های پیچیده، مثل روتینهای آلاینده یا روتین های رمزنگاری رها می کند. تشخیص بسیاری از توالی های دستورها در این صورت مطلوب خواهد بود. در حقیقت، یک آنالیزر بایستی قادر به ترسیم کل کدهای مضر (که قبلا شناخته شده اند و راجع به آنها مطالبی نوشته شده است) باشد تا در صورت روبرو شدن با آنها در خلال آنالیز، بتوان بسرعت عاملیت کد را تشخیص داد! با در نظر داشتن این مفاهیم، ما قصد داریم که یک چارچوب (framework) را برای نگهداری مجموعه ای از الگوهای کدنویسی معمول پیاده سازی کرده و آنرا در کنار PEAT قرار دهیم. شاید پروژه هایی را برای تشخیص کدها و نرم افزارهای مضر تقبل کنید. در این گونه موارد بسیاری از افراد از ابزارهایی مانند IDA Pro

[7] section alignment padding

یا OllyDBG یا Fenris و ... جهت انجام بررسی های خود استفاده می کنند. اما می توان گفت پیاده سازی فعلی از PEAT به عنوان یک زیرمجموعه مفید برای ابزارهای یادشده عمل می کند. برای مثال، IDA Pro کاملا برای آنالیز کد مناسب است، اما تاثیر و میزان کارایی آن در کنار PEAT دو چندان می شود. در این حالت، آنالیزر نیاز به آنالیز داده های اسمبلی مربوط به کل برنامه ندارد و تنها روی نواحی تعیین شده (نواحی مشکوک) توسط PEAT کار خواهد کرد، به این ترتیب زمان آنالیز به مراتب کاهش می یابد.

ترجمه: سعید بیکی (cephexin@secumania.net)

Secumania Security & Vulnerability Research Lab
www.secumania.net