Archive of SID

مهندسی معکوس کد دودویی نرم‌افزارهای شبکه برای شناسایی ساختار و معنای پیام‌های پروتکل

نیره مؤمنیان[۱]* و بهروز ترک لادانی[۲]

[۱] مجتمع فناوری اطلاعات، ارتباطات و امنیت، دانشگاه صنعتی مالک اشتر، تهران، ایران

[۲] گروه کامپیوتر، دانشکده فنی و مهندسی، دانشگاه اصفهان، اصفهان، ایران

چکیده

مهندسی معکوس برنامه‌های کاربردی شبکه، به‌خصوص از دیدگاه امنیت، بسیار مورد توجه قرار گرفته و اهمیت بالایی دارد. بسیاری از برنامه‌های کاربردی شبکه، از پروتکل‌های خاصی را که ویژگی‌های آنها برای عموم در دسترس نیست، استفاده می‌کنند. مهندسی معکوس این برنامه‌های کاربردی، می‌تواند اطلاعات مورد نیاز برای فهم پروتکل‌های ناشناخته مستقر در آنها فراهم کند؛ دسترسی به این اطلاعات می‌تواند بسیاری از وظایف، از جمله بازبینی عمیق پروتکل در نسل جدید دیواره‌های آتش و تحلیل کدهای دودویی مشکوک را تسهیل کند. با این وجود، اگرچه پژوهش‌های بسیاری در این زمینه انجام شده، اما این پژوهش‌ها در بیش‌تر موارد فقط بر استخراج ساختار نحوی پیام‌های پروتکل متمرکز شده‌اند. در این مقاله، روش‌های جدیدی برای بهبود استخراج ساختار نحوی و معنایی پیام‌های پروتکل از طریق مهندسی معکوس کد دودویی برنامه‌های کاربردی شبکه ارائه شده است. برای این کار، از ترکیب تحلیل پویا و ایستای کدهای دودویی استفاده می‌شود. به‌منظور ارزیابی روش‌های پیشنهادی، چهار پروتکل مختلف لایه کاربرد شامل DNS، eDonkey، Modbus و Stun تحلیل شده است. نتیجه آزمایش‌ها نشان می‌دهد که روش‌های پیشنهادی، نه‌تنها می‌توانند ساختار نحوی پیام را کامل‌تر از روش‌های مشابه استخراج کنند، بلکه معانی سودمندی از پیام‌های پروتکل نیز استخراج می‌کنند که در روش‌های قبلی قابل دست‌یابی نیست.

واژگان کلیدی: مهندسی معکوس، استخراج فرمت پیام، اطلاعات معنایی.

Reverse Engineering of Network Software Binary Codes for Identification of Syntax and Semantics of Protocol Messages

Nayereh Momenian*[1] & Behrouz Tork Ladani[2]

[1]Malek-Ashtar University of Technology, Tehran, Iran

[2]Department of Computer Engineering, Engineering Faculty, Isfahan University, Isfahan, Iran

Abstract

Reverse engineering of network applications especially from the security point of view is of high importance and interest. Many network applications use proprietary protocols which specifications are not publicly available. Reverse engineering of such applications could provide us with vital information to understand their embedded unknown protocols. This could facilitate many tasks including deep protocol inspection in next generation firewalls and analysis of suspicious binary codes .

The goal of protocol reverse engineering is to extract the protocol format and the protocol state machine. The protocol format describes the structure of all messages in protocol and the protocol state machine describes the sequence of messages that the protocol accept. Recently, there has been rising interest in automatic protocol reverse engineering. These works are divided into activities that extract protocol format and activities that extract protocol state machine. They can also be divided into those uses as input network traffic and those uses as input program implements the protocol. However, although there are some researches in this field, they mostly focused on extracting syntactic structure of the protocol messages.

* Corresponding author

*نویسندهٔ عهده‌دار مکاتبات

● تاریخ ارسال: ۹۶/۰۷/۱۷ ● تاریخ پذیرش: ۹۸/۰۴/۰۹ ● نوع مطالعه: پژوهشی

www.SID.ir

Archive of SID

In this paper, some new techniques are presented to improve extracting the format (both the syntax and semantics) of protocol messages via reverse engineering of binary codes of network applications. To do the research, an integration of dynamic and static binary code analysis are used. The field extraction approach first detects length fields and separators and then by applying rules based on compiler principles locates all the fields in the messages. The semantic extraction approach is based on the semantic information available in the program implements of the protocol and also information exists in the environment of the program.

For evaluating the proposed approach, four different network applications including DNS, eDonkey, Modbus, and STUN were analyzed. Experimental results show that the proposed techniques not only could extract more complete syntactic structure of messages than similar works, but also it could extract a set of advantageous semantic information about the protocol messages that are not achievable in previous works.

Keywords: Reverse engineering, Protocol format extraction, Semantic information.

۱- مقدمه

مهندسی معکوس پروتکل‌های لایه کاربرد[1]، کاربردهای بسیاری به‌خصوص در زمینه امنیت شبکه دارند؛ زیرا امروزه تهدیدهای امنیتی تکامل‌یافته و حمله‌های شبکه بر لایه کاربرد متمرکز شده‌اند؛ برای جلوگیری از این حمله‌ها، محصولات امنیتی به مشخصات پروتکل‌های مورد استفاده در برنامه‌های کاربردی شبکه[2] نیاز دارند. برای نمونه، با توجه به این که بسیاری از حمله‌ها از آسیب‌پذیری‌های[3] موجود در برنامه‌ها استفاده می‌کنند، برای جلوگیری از آنها باید آسیب‌پذیری‌ها را شناسایی کرد. این کار با فراهم‌کردن مشخصات پروتکل‌ها برای فازرهای هوشمند[4]، به‌منظور تولید ورودی‌های ساخت‌یافته[5] نامعتبر، تسهیل می‌شود. با این وجود، بسیاری از برنامه‌های کاربردی شبکه از پروتکل‌های خاصی که ویژگی‌های آنها برای عموم منتشر نشده، استفاده می‌کنند. همچنین شبکه‌های بات[6] که یکی از عوامل مهم حمله‌های DoS[7] در اینترنت هستند، از پروتکل‌های مخصوص به خود برای کانال‌های فرماندهی و کنترل[8] استفاده می‌کنند؛ بنابراین، مهندسی معکوس پروتکل برای استخراج مشخصات پروتکل‌های ناشناخته مورد نیاز است.

هدف از مهندسی معکوس پروتکل، استخراج فرمت پیام‌های مبادله شده و ماشین حالت پروتکل است. فرمت پیام پروتکل، ترتیب مشخصی از فیلدهای با معانی مختلف را نشان می‌دهد که هر یک، محدوده خاص خود را دارند. معنای فیلد[9] مشخص می‌کند چه نوع اطلاعاتی در فیلد قرار گرفته است، برای مثال نام فایل. محدوده فیلد[10]، مکان ابتدا و انتهای فیلد را مشخص می‌کند. ماشین حالت پروتکل، دنباله پیام‌های معتبر پروتکل را تعیین می‌کند؛ به عبارت دیگر، ترتیب ارسال پیام‌های با نوع مختلف در ماشین حالت پروتکل تعیین می‌شود؛ بنابراین برای استخراج ماشین حالت پروتکل، باید انواع مختلف پیام‌های پروتکل، از هم تمایز داده شوند. در این مقاله، بر استخراج فرمت پیام‌های پروتکل متمرکز می‌شویم زیرا پیش‌نیاز استخراج ماشین حالت پروتکل است و به‌علاوه همان‌طور که خواهیم دید، هنوز چالش‌های زیادی در انجام این کار وجود دارد.

مطالعات به‌نسبه زیادی در زمینه مهندسی معکوس پروتکل‌ها انجام شده است. این مطالعات از دو دیدگاه مبتنی بر شبکه[11] و مبتنی بر برنامه[12] استفاده می‌کنند. در دیدگاه مبتنی بر شبکه، منبع اطلاعاتی مورد استفاده، ترافیک شبکه است. در این دیدگاه، برای استخراج مشخصات پروتکل، از ویژگی‌های مجموعه پیام‌های مبادله‌شده در شبکه استفاده می‌شود. این دیدگاه، محدود به تنوع پیام‌های مورد تحلیل است و دقت کمی دارد؛ به‌علاوه به‌دلیل کمبود معنا در ترافیک شبکه، اطلاعات محدودی از پروتکل استخراج می‌شود. در دیدگاه مبتنی بر برنامه، علاوه‌بر ترافیک شبکه از برنامه پیاده‌ساز پروتکل به‌عنوان منبع غنی اطلاعات استفاده می‌شود. این دیدگاه، با نظارت بر چگونگی پردازش پیام ورودی توسط برنامه پیاده‌ساز پروتکل، سعی در استخراج ویژگی‌های پروتکل دارد. در مقایسه با دیدگاه مبتنی بر شبکه، این دیدگاه از دقت بیش‌تری برخوردار است و می‌تواند ویژگی‌های کامل‌تری از پروتکل استخراج کند؛ زیرا برنامه پیاده‌ساز پروتکل اطلاعات ارزشمندی از پروتکل در اختیار دارد. تاکنون، مطالعات زیادی

[1] Application Layer
[2] Network Applications
[3] Vulnerability
[4] Smart Fuzzer
[5] Constructed Input
[6] Botnets
[7] Denial Of Service
[8] Command and Control (C&C)
[9] Field Semantic
[10] Field Boundary
[11] Network Based
[12] Program Based

www.SID.ir

Archive of SID

با استفاده از هر دو دیدگاه انجام شده است؛ این مطالعات به‌طور کلی بر استخراج ساختار نحوی پیام متمرکز هستند و معانی بسیار محدودی از پیام‌های پروتکل استخراج می‌کنند.

در این مقاله قصد داریم با ترکیب تحلیل پویا و ایستای برنامه پیاده‌ساز پروتکل، سامانه جدیدی برای استخراج کامل‌تر فرمت پیام‌ها ارائه دهیم. سامانه ارائه شده می‌تواند معانی سودمندی از فیلدهای پیام استخراج کند؛ برای این کار، بیش‌تر از اطلاعات موجود در برنامه که برای ارتباط با کاربر فراهم شده است، استفاده می‌شود.

به‌طور کلی در این مقاله، ایده‌های جدید زیر ارائه شده است:

- ✓ **روش‌های جدیدی برای شناسایی فیلد طول:** فیلد طول، برای مشخص‌کردن اندازه قسمتی از پیام با طول متغیر استفاده می‌شود. در پژوهش‌های پیشین، فنونی برای شناسایی این فیلد ارائه شده است، اما این فنون نمی‌توانند همه انواع فیلدهای طول را شناسایی کند. در این مقاله، فنون جدیدی برای شناسایی کامل‌تر فیلد طول ارائه می‌دهیم.
- ✓ **روشی جدید برای مشخص‌کردن محدوده فیلدها:** تاکنون، پژوهش‌های زیادی برای مشخص‌کردن محدوده فیلدها انجام شده، با این وجود روش‌های ارائه‌شده با مشکلاتی در تشخیص دقیق محدوده فیلدها مواجه هستند و محدوده برخی فیلدها را به‌اشتباه، کوچک‌تر یا بزرگ‌تر شناسایی می‌کنند. در این مقاله، شیوه جدیدی برای استخراج دقیق‌تر محدوده فیلدها ارائه می‌دهیم.
- ✓ **روش جدیدی برای شناسایی فیلد نوع:** فیلد نوع، برای تشخیص طبقه‌های مختلف یک شیء به‌کار می‌رود و در بسیاری از پروتکل‌ها مورد استفاده قرار می‌گیرد. در این مقاله، شیوه‌ای جدید برای تشخیص فیلد نوع ارائه می‌دهیم که می‌تواند سایر مقادیر معتبر فیلد نوع را که در پیام ظاهر نشده‌اند نیز شناسایی نماید. نتایج این روش می‌تواند در بسیاری از سامانه‌های امنیتی مانند سامانه‌های تشخیص نفوذ مورد استفاده قرار گیرد.
- ✓ **روشی جدید برای استخراج معانی فیلدها:** معانی فیلدهای پیام، برای فهم هدف پیام و پروتکل مورد نیاز هستند. به‌علاوه بسیاری از کاربردهای امنیتی برای انجام وظایف خود به معانی فیلدها نیاز دارند. بیش‌تر روش‌های پیشین معانی بسیار محدود و از پیش تعیین شده‌ای از فیلدها استخراج می‌کنند؛ مانند طول و شماره درگاه. این روش‌ها در صورتی که فیلدی معنایی به جز معانی از پیش تعیین شده داشته باشد، نمی‌توانند آن‌ها را شناسایی کنند. تنها روشی که برای شناسایی معانی از پیش تعیین‌نشده فیلدها پیشنهاد شده، روشی است که Dispatcher [25] از آن استفاده می‌کند و در آن از اطلاعات پیش‌نمونه[1] توابع شناخته‌شده استفاده می‌شود. این روش محدودیت‌هایی دارد، از جمله اینکه محدود به دردسترس‌بودن پیش نمونه توابع است. بنابراین، روش‌های دیگری برای شناسایی کامل‌تر معانی فیلدهای پیام مورد نیاز است. در این مقاله، روشی جدید برای استخراج معانی فیلدها ارائه شده که در آن از اطلاعاتی که برای ارتباط با کاربر در برنامه موجود است، استفاده می‌شود.

برای ارزیابی سامانهٔ پیشنهادی، برنامه‌های اجرایی دودویی چهار پروتکل DNS، eDonkey، Modbus و STUN را تحلیل کردیم. نتایج ارزیابی نشان می‌دهد که روش پیشنهادی در استخراج فرمت پیام، کامل‌تر و دقیق‌تر از کارهای پیشین است و می‌تواند معانی سودمندی از فیلدهای پیام استخراج کند که در روش‌های قبلی قابل استحصال نیست.

در ادامه مقاله، در بخش ۲ کارهایی که در این زمینه انجام شده مرور می‌شود. پس از آن ابتدا در بخش ۳ دیدگاه و معماری سامانه پیشنهادی و سپس در بخش ۴ روش‌های پیشنهادی برای استخراج فرمت پیام ارائه می‌شود. جزئیات پیاده‌سازی و روش ارزیابی مورد استفاده در بخش ۵ و در پایان در بخش ۶ نتیجه‌گیری مقاله ارائه شده است.

۲- کارهای گذشته

نخستین فعالیت‌ها در زمینه مهندسی معکوس پروتکل، دشوار و زمان‌بر بودند و با مداخله یک متخصص به شیوه دستی انجام می‌شدند. برای مثال، مهندسی معکوس پروتکل پیام‌رسان یاهو[2] چندین‌سال زمان صرف کرده است؛ برای حل این مشکل و انجام مهندسی معکوس پروتکل در مدت زمانی مناسب، مطالعات گسترده‌ای انجام شده که پروژه PI[3] [26] یکی از نخستین آنها است. این پژوهش از دیدگاه مبتنی بر شبکه استفاده می‌کند و برای نخستین‌بار، ایده استفاده از الگوریتم‌های هم‌ترازی دنباله‌ها[4] را، به‌منظور یافتن مشابهت میان پیام‌های یک پروتکل، پیشنهاد کرده و از این طریق، توانسته است محدوده برخی فیلدها را شناسایی کند. الگوریتم‌های هم‌ترازی دنباله‌ها، به‌طور گسترده‌ای در بیولوژی برای یافتن روابط میان دو دنباله از اطلاعات ژنتیکی استفاده

[1] Prototype
[2] Yahoo Messenger
[3] Protocol Informatics Project
[4] Sequence Alignment

www.SID.ir

Archive of SID

می‌شوند. بسیاری از پژوهش‌ها مبتنی بر شبکه از شیوهٔ ارائه‌شده در PI استفاده و سعی کرده‌اند آن را بهبود دهند. Discoverer [27] از خوشه‌بندی بازگشتی و هم‌ترازی مبتنی بر نوع به جای هم‌ترازی مبتنی بر بایت استفاده کرده است. ProDecoder [28] با استفاده از n-gram‌ها، واژه‌های کلیدی[1] پروتکل را استخراج و پیام‌ها را بر اساس واژه‌های کلیدی، خوشه‌بندی می‌کند؛ سپس هم‌ترازی را به پیام‌های هر خوشه اعمال می‌کند. *فن پن*[2] و همکاران برای کاهش مقیاس هم‌ترازی، پیشنهاد استفاده از هم‌ترازی چندگانه متوالی را ارائه کردند [29]. Netzob [30] پیام‌ها را بر اساس عملیاتی که برای انجام آن مبادله شده‌اند، خوشه‌بندی می‌کند و برای هم‌ترازی پیام‌های هر خوشه از اطلاعات وضعیتی و محیطی[3] موجود در پیام‌های مبادله‌شده مانند نام میزبان، نام کاربری و شماره درگاه استفاده می‌کند. [7] سعی بر بهبود روش عملکردی Netzob کرده است. به این منظور برای تشخیص نوع پیام‌ها از مدل تخصیص پنهان دیریکله[4] استفاده می‌کند. بر خلاف سایر روش‌های یادشده که از مقایسه انواع پیام‌های یک پروتکل با هم سعی در استخراج ساختار پیام دارند، NEMESYS [8] از بررسی میزان مشابهت مقادیر مختلف درون یک پیام برای استخراج ساختار آن استفاده می‌کند. از شیوه‌های آماری و داده‌کاوی نیز برای استخراج مشخصات پیام پروتکل‌ها با استفاده از ترافیک شبکه استفاده شده است [9-11].

پژوهش‌های یادشده، همگی سعی در استخراج ویژگی‌های پروتکل از ترافیک شبکه داشتند؛ اما دسته دیگری از پژوهش‌ها هستند که از برنامه پیاده‌ساز پروتکل به‌عنوان منبع اطلاعاتی استفاده می‌کنند. Polyglot [36] نخستین پژوهش انجام‌شده از این دسته است که بر چگونگی پردازش پیام ورودی، توسط برنامه پیاده‌ساز پروتکل نظارت می‌کند. در این پژوهش، روش‌هایی برای شناسایی جداساز[5]‌ها، واژه‌های کلیدی، فیلد طول و اشاره‌گر از طریق مشاهده شیوه پردازش آنها توسط دستورهای برنامه ارائه شده است. این پژوهش، برای شناسایی محدوده فیلدها، از این استدلال استفاده می‌کند که هر فیلد، یک واحد معنایی است و برای پردازش آن از دستورهای یکسانی استفاده می‌شود. بنابراین بایت‌های مجاوری را که با هم به‌عنوان عمل‌وند دستورهای CPU قرار می‌گیرند، به‌عنوان یک فیلد شناسایی می‌کند. با توجه به این که طول هر عمل‌وند می‌تواند درنهایت به اندازه طول کلمه سامانه، باشد این روش محدوده فیلدهای طولانی‌تر از اندازه کلمه سامانه را به‌درستی تشخیص نمی‌دهد. Autoformat [37] کانتکست[6] اجرایی را در زمان پردازش پیام ورودی توسط برنامه جمع‌آوری می‌کند و بایت‌های مجاوری که توسط توابعی یکسان، مورد دسترسی قرار گرفته‌اند، به‌عنوان یک فیلد در نظر می‌گیرد. درصورتی‌که محدوده یک فیلد، زیرمجموعه محدوده فیلدی دیگر باشد، رابطه پدر و فرزندی میان این دو فیلد تشکیل می‌دهد و به این ترتیب، ساختار سلسله‌مراتبی میان فیلدها را استخراج می‌کند. اگرچه با استفاده از این شیوه در برخی موارد نتیجه صحیح به‌دست می‌آید، اما به‌دلایلی از جمله اینکه ممکن است، چندین فیلد مختلف توسط یک تابع پردازش شوند، احتمال وقوع خطا در آن زیاد است. Dispatcher فنونی برای استخراج فرمت پیام‌های ارسالی با استفاده از بازسازی میان‌گیر[7] خروجی پیشنهاد داده است. به‌علاوه، از اطلاعات موجود در پیش‌نمونه توابع برای فهم معانی فیلدها استفاده می‌کند و معنای فیلد را زمانی که به‌عنوان آرگومان تابعی شناخته‌شده استفاده شود، شناسایی می‌کند. شناسایی معانی با استفاده از این روش محدود به در دسترس‌بودن اطلاعات پیش‌نمونه توابع است، درحالی‌که بسیاری از توابع استفاده‌شده در برنامه، توسط کاربر تعریف شده‌اند و اطلاعات پیش‌نمونه آنها قابل دسترس نیست؛ به‌علاوه بسیاری از فیلدها به‌عنوان آرگومان هیچ تابعی مورد استفاده قرار نمی‌گیرند. Icefex [38] شیوه شناسایی محدوده فیلدها در Polyglot را با استدلال زیر بهبود می‌دهد: به‌طورمعمول برنامه پیاده‌ساز پروتکل، پیام را در مراحل مختلفی مانند واژه‌شناسی[8]، تجزیه[9] و ارزیابی[10] پردازش می‌کند؛ دستورهای مرحله واژه‌شناسی و تجزیه، پیام را بدون توجه به محدوده فیلدها پردازش می‌کنند؛ اما دستورهای مرحله ارزیابی محکم‌ترین مدرک برای شناسایی محدوده فیلدها هستند. بر اساس این استدلال، در این پژوهش برای شناسایی محدوده فیلدها عمل‌وندهای دستورهای مرحله ارزیابی مورد بررسی قرار می‌گیرند.

مطالعات بررسی‌شدهٔ بالا، پروتکل را با هدف استخراج فرمت پیام مهندسی معکوس کرده‌اند. بررسی و مقایسه روش‌های مبتنی بر شبکه نشان می‌دهد که درکل این روش‌ها برای استخراج ساختار نحوی پیام از الگوریتم‌های هم‌ترازی

---

[1] Keyword
[2] Fan Pan
[3] Contextual and environmental information
[4] Latent Dirichlet Allocation (LDA)
[5] Separator
[6] Context
[7] Buffer Deconstruction
[8] Lexing
[9] Parsing
[10] Evaluating

www.SID.ir

Archive of SID

استفاده می‌کنند و با استخراج قسمت‌های مشابه پیام‌ها، محدوده هر فیلد را از فیلدهای دیگر تمایز می‌دهند. دقت این روش‌ها به تنوع پیام‌های مبادله‌شده در شبکه وابستگی شدیدی دارد؛ به‌علاوه اگرچه این روش‌ها محدوده برخی فیلدها در پروتکل‌های متنی را به‌درستی تشخیص می‌دهند؛ اما در پروتکل‌های باینری-کد شده[1] به دلیل تعداد کم نویسه‌های مورد استفاده برای کدگزاری داده و افزایش احتمال تشابه نویسه‌ها، دقت و کارایی به‌شدت کاهش می‌یابد. در روش‌های مبتنی بر شبکه معنای تعداد محدودی از فیلدها مانند فیلد طول، شماره درگاه، نام فایل و غیره بر اساس ویژگی‌های آماری، وابستگی میان فیلدهای مختلف پیام‌ها و اطلاعات وضعیتی و محیطی پیام تشخیص داده می‌شود که نتیجه حاصل دقت بسیار کمی دارد. از طرف دیگر بررسی و مقایسه روش‌های مبتنی بر برنامه نشان می‌دهد که این روش‌ها برای استخراج فرمت پیام بر شیوه پردازش پیام توسط برنامه نظارت می‌کنند. به این ترتیب با شناسایی معانی فیلدهای طول و جداساز و بررسی بایت‌هایی که به‌عنوان عمل‌وند دستورهای CPU قرار می‌گیرند، محدوده فیلدها را شناسایی می‌کنند. دقت این روش‌ها، وابستگی کمی به تنوع پیام‌های مورد تحلیل دارد و در برخی موارد حتی با داشتن یک پیام نیز می‌توانند نتایج قابل قبولی ارائه دهند؛ البته با توجه به این که برخی دستورهای برنامه، داده‌ها را بدون توجه به محدوده آن‌ها پردازش می‌کنند و این که فیلدهای طول در همه موارد شناسایی نمی‌شوند، نیاز است روش‌های پیشین برای ارائه نتیجه‌ای کامل‌تر و صحیح‌تر اصلاح شوند. علاوه‌بر این، محدوده شناسایی‌شده برای فیلدها، محدود به تعداد بایت قابل استفاده در عمل‌وند دستورهای CPU است که درنهایت می‌تواند به اندازه کلمه سامانه باشد، بنابراین محدوده فیلدهای طولانی‌تر از این مقدار توسط این روش‌ها قابل شناسایی نیست. روش‌های مبتنی بر برنامه برای شناسایی معانی فیلدها بر دستورهایی که فیلدها را به‌عنوان ورودی توابع شناخته‌شده استفاده می‌کنند، نظارت می‌کنند. به این ترتیب معانی برخی فیلدها شناسایی می‌شود که البته نتایج محدود به شناخته‌شده‌بودن تابع است. همان‌طور که مشخص است، مطالعات انجام‌شده در هر دو روش مبتنی بر شبکه و مبتنی بر برنامه در استخراج ساختار نحوی و معنایی پیام‌ها کاستی‌هایی دارند. منشأ وجود این کاستی‌ها در روش‌های مبتنی بر شبکه، نبودن معنا در ترافیک شبکه است؛ اما در روش‌های مبتنی بر برنامه با توجه به معانی ارزشمند موجود در برنامه پیاده‌ساز پروتکل، از دلایل این کاستی‌ها می‌توان به عدم استفاده از مزایای روش تحلیل ایستا در کنار تحلیل پویای کد دودویی اشاره کرد [12].

در این مقاله برای رفع کاستی‌ها و مشکلات موجود با استفاده از مزایای ترکیب شیوه تحلیل ایستا و پویای کد دودویی برنامه پیاده‌ساز پروتکل، روش‌های جدیدی برای استخراج فرمت پیام ارائه شده تا علاوه‌بر استخراج محدوده فیلدها با دقت بیش‌تر و به‌طور کامل‌تر، معانی بیشتری از فیلدهای پیام نیز شناسایی شود.

۳- رویکرد و معماری سامانهٔ پیشنهادی

در این بخش، رویکرد مورد استفاده و معماری سامانهٔ پیشنهادی خود را معرفی می‌کنیم. ما از دیدگاه مبتنی بر برنامه، برای مهندسی معکوس پروتکل استفاده می‌کنیم، یعنی منبع اطلاعاتی ما برنامه پیاده‌ساز پروتکل است. به‌دلیل آن که متن برنامه[2] برنامه پیاده‌ساز پروتکل در دسترس نیست و فقط به فایل اجرایی برنامه دسترسی داریم که از نوع دودویی و حاوی دستورهای زبان ماشین است، نیاز است از روش‌های مهندسی معکوس دودویی استفاده شود. روش‌های پیشین مبتنی بر برنامه، در بیش‌تر موارد از تحلیل پویای دودویی برای این کار استفاده کرده‌اند. با وجود این که استفاده از تحلیل پویا و نظارت بر اجرای برنامه پیاده‌ساز پروتکل در هنگام پردازش پیام، اطلاعات ارزشمندی در مورد فرمت پیام فراهم می‌کند، اما این اطلاعات، محدود به یک مسیر اجرایی از برنامه است. روش‌های تحلیل ایستا، امکان مشاهده همه مسیرهای اجرایی برنامه را فراهم می‌کنند. بنابراین در این مقاله برای استخراج دقیق‌تر و کامل‌تر ویژگی‌های پروتکل، از ترکیب تحلیل پویا و ایستای دودویی برنامه پیاده‌ساز پروتکل استفاده شده است.

شکل (۱) معماری سامانهٔ پیشنهادی را نمایش می‌دهد. سامانهٔ پیشنهادی از سه مؤلفه تشکیل شده است: مؤلفهٔ تحلیل پویا، مؤلفه تحلیل ایستا و مؤلفه استخراج فرمت. این مؤلفه‌ها از ماژول‌هایی تشکیل شده‌اند که ابزارهای مورد استفاده برای پیاده‌سازی برخی از آنها در زیر هر ماژول نشان داده شده است. در بخش پنجم این ابزارها همراه با جزئیات بیش‌تر توصیف خواهند شد. مؤلفه تحلیل پویا، برنامه پیاده‌ساز پروتکل و پیام ورودی شبکه را به‌عنوان ورودی دریافت می‌کند. این مؤلفه از دو ماژول نظارت بر اجرا[3] و استخراج

[1] Binary-Coded Protocol

[2] Source Code

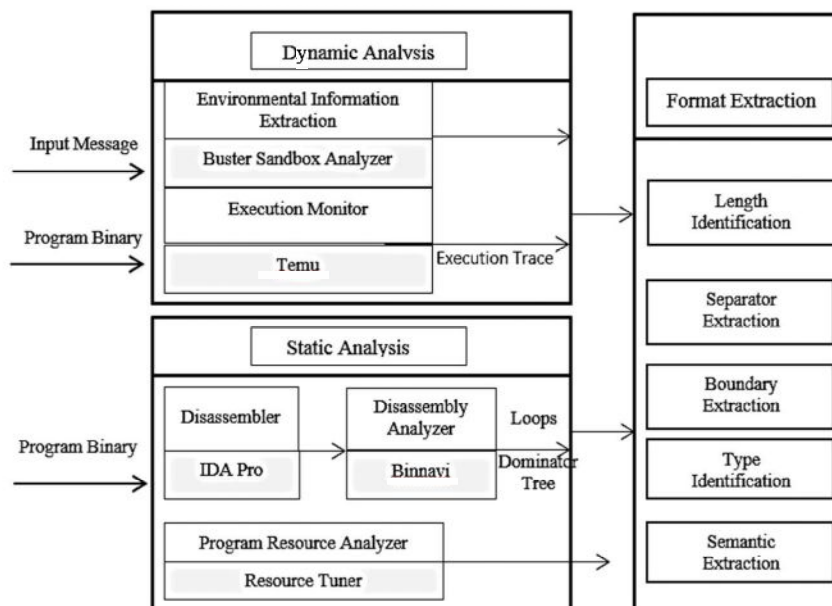
[3] Execution Monitor

www.SID.ir

Archive of SID

اطلاعات محیطی[1] تشکیل شده است. ماژول نظارت بر اجرا، برنامه را در یک محیط شبیه‌سازی‌شده اجرا و بر چگونگی پردازش پیام ورودی توسط برنامه نظارت می‌کند؛ برای این کار از تحلیل پویای آلودگی[2] استفاده می‌کند و هنگام دریافت داده ورودی از شبکه آن را آلوده می‌کند؛ اطلاعات آلودگی، با اجرای دستورهایی که داده‌های آلوده‌شده را از مبدأ به مقصد منتقل می‌کنند (مانند عملیات انتقالی، محاسباتی یا منطقی) انتشار می‌یابند. این ماژول، دستورهای مورد استفاده توسط برنامه برای پردازش داده‌های آلوده‌شده را ضبط[3] می‌کند و خروجی آن، یک ردّ اجرایی[4] است که شامل اطلاعات همه دستورهای پردازش داده‌های آلوده، مقادیر عملوندها و اطلاعات آلودگی آنها است. ماژول استخراج اطلاعات محیطی برنامه، در زمان اجرای برنامه اطلاعاتی که برای ایجاد ارتباط و هماهنگی میان دو طرف استفاده می‌شود (مانند اطلاعات مورد نیاز برای احراز هویت کاربر) و نیز اطلاعات منابعی که در زمان پردازش دسترسی می‌شوند (مانند فایل‌های ایجاد یا ویرایش‌شده) جمع‌آوری می‌کند. مؤلفه تحلیل ایستا، از سه ماژول دیس‌اسمبلر[5]، تحلیل‌گر دیس‌اسمبلی[6] و تحلیل‌گر منابع برنامه[7] تشکیل شده است. ورودی این مؤلفه، دودویی برنامه پیاده‌ساز پروتکل است. از ماژول دیس‌اسمبلر، برای تبدیل کد زبان ماشین به کد زبان اسمبلی استفاده می‌شود؛ سپس تحلیل‌گر دیس‌اسمبلی با تحلیل کد دیس‌اسمبلی تولیدشده، حلقه‌های موجود در کد را استخراج و گراف جریان اجرا[8] و درخت تسلط[9] را تولید می‌کند. ماژول تحلیل‌گر منابع برنامه، وظیفه استخراج اطلاعات منابع استفاده‌شده در برنامه مانند واسط گرافیکی کاربر و پایگاه داده‌ها را بر عهده دارد.

در مؤلفه استخراج فرمت، خروجی‌های دو مؤلفه دیگر به‌عنوان ورودی داده می‌شوند و با استفاده از پنج ماژول، ساختار نحوی و معنایی پیام استخراج می‌شود. جزئیات این ماژول‌ها در بخش ۴ توصیف خواهند شد.



(شکل-۱): معماری سامانهٔ پیشنهادی

(Figure-1): The Proposed System Architecture

۴- استخراج فرمت پیام

فرمت پیام پروتکل، مجموعه‌ای از فیلدها با معانی مختلف را نشان می‌دهد که با ترتیب خاصی در کنار هم قرار گرفته‌اند؛ هر فیلد محدوده خاص خود را دارد و می‌تواند طولی ثابت یا متغیر داشته باشد. در این بخش، ابتدا با ارائه فنونی برای شناسایی فیلد طول و جداسازها، محدوده فیلدهای با طول متغیر شناسایی می‌شود؛ سپس شیوه پیشنهادی، محدوده فیلدهای با اندازه ثابت را نیز شناسایی می‌کند. در ادامه، برای شناسایی معانی فیلدها، ابتدا روشی برای شناسایی فیلد نوع و سپس روشی برای استخراج معانی از فیلدهای پیام ارائه می‌شود.

[1] Environmental Information Extraction
[2] Dynamic Taint Analysis
[3] Capture
[4] Execution Trace

[5] Disassembler
[6] Disassembly Analyzer
[7] Program Resource Analyzer
[8] Control Flow Graph
[9] Dominator Tree

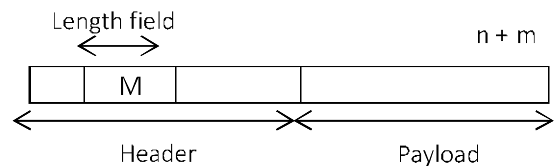
www.SID.ir

Archive of SID

۴-۱- شناسایی فیلد طول

فیلد طول، برای مشخص‌کردن طول قسمتی از پیام با اندازه متغیر که پس از آن می‌آید (فیلد هدف[1])، استفاده می‌شود. پژوهش‌های پیشین بر پایه این استدلال بودند که برنامه از فیلد طول برای محاسبه انتهای فیلد هدف استفاده می‌کند. دو روش برای انجام این کار وجود دارد: نخست، افزودن فیلد طول به اشاره‌گر ابتدای فیلد هدف. دوم، استفاده از یک حلقه با شرط توقف مبتنی بر فیلد طول. برای شناسایی فیلد طول باید دستورهایی که این دو روش را پیاده‌سازی می‌کنند شناسایی کرد. کارهای پیشین [36]، برای شناسایی دستورهای پیاده‌ساز حلقه، کدهای مکرری که در آنها شمارنده، شرط توقف و دستور پرش به عقب دیده می‌شود، استخراج می‌کنند. اگرچه با استفاده از این علائم، بیش‌تر حلقه‌های موجود در برنامه شناسایی می‌شوند، اما باید توجه کرد که برخی از دستورها به‌طور ضمنی حلقه ایجاد می‌کنند؛ برای نمونه، دستورهای اسمبلی که دارای پیشوند rep هستند به تعداد مشخص‌شده در رجیستر ecx تکرار می‌شوند؛ اما شرط توقف در آنها به‌وضوح دیده نمی‌شود؛ همچنین دستور پرش به عقب در آنها وجود ندارد. بنابراین برای تشخیص کامل‌تر فیلد طول، لازم است علاوه‌بر استخراج حلقه‌های واضح، حلقه‌های ضمنی با شمارنده‌های آلوده‌شده را نیز شناسایی کرد.

اگرچه با شناسایی روش‌های محاسبه انتهای فیلد هدف، بسیاری از فیلدهای طول شناسایی می‌شوند، با این وجود با استفاده از این روش در برخی موارد، فیلد طول شناسایی نمی‌شود. روش پیشنهادی ما برای شناسایی کامل‌تر فیلد طول، مبتنی بر این استدلال است که در برخی برنامه‌ها، فیلد طول برای محاسبه انتهای فیلد هدف استفاده نمی‌شود زیرا انتهای فیلد هدف برای برنامه مشخص است و برنامه از فیلد طول برای بررسی صحیح بودن تعداد بایت‌های دریافتی از شبکه استفاده می‌کند. در این موارد، فیلد طول در محاسبه‌ای استفاده می‌شود که نتیجه آن با طولی معلوم، مانند طول کلی پیام یا مقدار فیلد طول دیگری در پیام، مقایسه خواهد شد.



(شکل-۲): نمونه‌ای از فیلد طول در سرآیند پیام
(Figure-2): The payload length filed example

یکی از مثال‌های بارز این مورد که در شکل (۲) نمایش داده شده، فیلد طول مورد استفاده در سرآیند[2] پیام است. به این ترتیب با استفاده از روش پیشنهادی، طول قسمت سرآیند پیام نیز شناسایی می‌شود.

۴-۲- شناسایی جداسازها

جداسازها فیلدهایی هستند که برای تفکیک محدوده قسمت‌های با طول متغیر یک پیام استفاده می‌شوند. برای مثال (/r/n) در پروتکل متنی HTTP، یک جداساز است. هر جداساز برای تفکیک محدوده بخشی از پیام استفاده می‌شود. برنامه پیاده‌ساز پروتکل برای تشخیص محدوده فیلدها باید مکان جداساز در داده ورودی را تشخیص دهد. برای این کار، بایت‌های مختلف دریافتی از شبکه را با مقادیر جداساز مقایسه می‌کند و هنگامی که مقایسه‌ای صحیح پیدا شد، محدوده فیلد را تعیین می‌کند. بنابراین برای شناسایی جداسازها، در ردّ اجرایی مقایسه‌های مقادیر ثابت و بایت‌های آلوده‌شده را جستجو می‌کنیم. البته همان‌طور که بعداً هم خواهیم دید همه این مقایسه‌ها نشان‌دهندهٔ جداساز نیست، آنچه که جداساز را از سایر موارد متمایز می‌کند، آن است که جداساز به‌طورتقریبی با همه بایت‌های متوالی قسمتی که مسئول تفکیک محدوده آن است مقایسه می‌شود. مفهوم مقایسه در اینجا گستره وسیعی از دستورها را در بر می‌گیرد که کمپایلر جهت تولید کد بهینه برای مقایسه از آنها استفاده می‌کند.

۴-۳- تشخیص محدوده فیلدها

هر یک از فیلدهای تشکیل‌دهنده پیام، محدوده‌ای دارند که مکان ابتدا و انتهای آنها را در پیام مشخص می‌کند و آنها را از سایر فیلدها جدا می‌کند. برنامه پیاده‌ساز پروتکل برای پردازش فیلدهای مختلف پیام باید مکان هر فیلد در پیام را بداند، تا بتواند فیلدها را پردازش کند. اگر فیلدی دارای اندازه‌ای ثابت باشد با مشخص‌بودن مکان ابتدای فیلد، برنامه همیشه مکان انتهای آن را می‌داند، اما درصورتی‌که اندازه فیلد متغیر باشد، مکان انتهای فیلد برای برنامه مشخص نیست و باید طبق روش‌هایی این مکان را تعیین کند. برای این کار در بیش‌تر موارد از نویسه‌های مشخصی جهت جداسازی فیلدها از هم استفاده و یا از فیلدی برای نگه‌داری طول استفاده می‌شود. بنابراین درصورتی‌که فیلدهای طول و جداسازها شناسایی شوند، محدوده فیلدهای با طول متغیر شناسایی می‌شود؛ اما برای جداسازی فیلدهای با طول ثابت از هم

[1] Target Field

[2] Header

www.SID.ir

Archive of SID

به‌ندرت از فیلدهای طول و جداساز استفاده می‌شود. برای شناسایی محدوده فیلدهای با طول ثابت باید به این نکته توجه کرد که هر فیلد یک واحد معنایی است و بایت‌هایی که متعلق به یک فیلد هستند، نیز یک معنای واحد دارند. بنابراین طبق استدلال ارائه‌شده در Polyglot دستورهای یکسانی آنها را پردازش می‌کنند. با این وجود باید به این نکته نیز توجه کرد که دستورهای سطح بالای نوشته‌شده، به‌منظور اجراشدن باید توسط کمپایلر به زبان ماشین ترجمه شوند؛ کمپایلر برای اجرای سریع‌تر دستورها توسط CPU آنها را بهینه‌سازی می‌کند؛ زیرا آنچه ماشین برای اجرا نیاز دارد، مجموعه دستورهایی ساده و سریع برای پردازش است. بنابراین ممکن است، دستورهای اسمبلی بدون توجه به محدوده فیلدها آنها را پردازش کنند؛ به‌علاوه تعداد بایت‌هایی که عمل‌وند دستورهای CPU می‌توانند داشته باشند، حداکثر به اندازه طول کلمه سیستم (چهار بایت برای سیستم ۳۲ بیتی و هشت بایت برای سیستم ۶۴ بیتی) است و اگر فیلدی طولانی‌تر از کلمه سیستم باشد، نیاز به چندین دستور برای پردازش آن است. به این ترتیب ممکن است حالاتی وجود داشته باشد که همه بایت‌هایی که در یک دستور پردازش می‌شوند، متعلق به یک فیلد نباشند یا اینکه تعدادی از بایت‌های یک فیلد را پردازش کنند.

در این مقاله برای تشخیص محدوده فیلدهای با اندازه ثابت از یک فرآیند چندمرحله‌ای استفاده می‌کنیم که در آن دستورهایی را که برای پردازش فیلدها بیش‌تر از سایر دستورها به محدوده فیلدها توجه می‌کنند در اولویت قرار دارند و در مراحل اولیه بررسی می‌کنیم. این فرایند چند مرحله‌ای به ترتیب اولویت در الگوریتم‌های ۱ تا ۴ نشان داده شده است. در هر الگوریتم مجموعه‌های θ و $\mathcal{F}$ ورودی‌ها و مجموعه $\mathcal{F}$ خروجی الگوریتم است. θ مجموعه همه دستورهای موجود در ردّ اجرایی برنامه است، به‌طوری‌که در هر دستور اطلاعات آلودگی[1] به‌اختصار در کنار هر رجیستر یا مکان حافظه نمایش داده شده است. به‌عنوان نمونه یکی از دستورهای این مجموعه mov %ebx(T1,0),%eax(T0) است. در این دستور T0 آلوده‌نبودن و T1 آلوده‌بودن بایت را نشان می‌دهد. پس از هر رجیستر یا مکان حافظهٔ آلوده، آفست بایت‌های آلوده نیز قرار می‌گیرد. بنابراین در این دستور بایت صفرم پیام ورودی در رجیستر ebx قرار گرفته است. $\mathcal{F}$ مجموعه فیلدهایی با محدوده شناسایی شده است که در ابتدا می‌تواند شامل فیلد طول، جداسازها و فیلدهای با طول متغیر باشد. هر فیلد به‌صورت یک دوتایی [x,y] نمایش داده می‌شود که در آن x آفست شروع فیلد و y آفست انتهای فیلد در پیام است.

در الگوریتم (۱)، عمل‌وند دستورهایی که برای تبدیل نوع داده‌ای بایت به نوع داده‌ای دیگری استفاده می‌شوند، مورد بررسی قرار می‌دهیم. دلیل بررسی عملوند این دستورها در مرحله نخست آن است که آنچه از شبکه دریافت می‌شود، جریانی از بایت‌ها است که برای پردازش آنها ابتدا باید با توجه به نوع و معنای فیلد به نوع داده‌ای مناسبی تبدیل شوند. بنابراین بایت‌هایی که در دستورهای تبدیل نوع مورد استفاده قرار می‌گیرند، متعلق به یک فیلد هستند. برای نمونه دستورهای اسمبلی زیر یکی از این دستورها هستند که برای تبدیل نوع داده‌ای بایت به نوع داده‌ای کلمه کاربرد دارند:

```
shl x, 8;   // shift left x by 8 byte.
add/or x, y
```

در ابتدا دستورهایی که برای تبدیل انواع داده‌ای استفاده می‌شوند، به‌صورت فهرستی در مجموعه $\mathcal{C}$ قرار می‌گیرند (خط ۱). در هر فهرست آفست بایت‌های استفاده‌شده به‌عنوان مقصد دستورها به‌عنوان ورودی تابع create_field داده می‌شود (خط ۴و۵). تابع create_field برای ایجاد یک فیلد، فهرستی از آفست بایت‌ها را به‌عنوان ورودی می‌گیرد و بایت‌های متوالی را در یک ساختار فیلد قرار می‌دهد. ساختار فیلد یک نقطه شروع و پایان در پیام را مشخص می‌کند؛ درنهایت فیلدهای ایجادشده در مجموعه $\mathcal{F}$ قرار می‌گیرند و این مجموعه به‌روز می‌شود.

	Algorithm 1
	Input: θ, $\mathcal{F}$
	Output: $\mathcal{F}$
	Begin
1:	$\mathcal{C} \leftarrow$ Set of instructions ϵ θ that are used to convert byte data type to another data type for tainted bytes.
2:	$d_o \leftarrow \phi$
3:	**for each** list $lst_i \in \mathcal{C}$
4:	$d_o \leftarrow$ offset of tainted bytes that are used as destination operand in all instructions of lst_i
5:	**Create_Field** (d_o)
6:	**Update** $\mathcal{F}$
	End

در مرحله دوم سعی داریم فیلدهای طولانی‌تر از اندازه کلمه سیستم را شناسایی کنیم. روش‌های پیشین، چنین فیلدهایی را به فیلدهایی با محدوده کوچکتر تقسیم می‌کردند. روش ما برای شناسایی این فیلدها بر اساس این استدلال است

[1] Taint

www.SID.ir

Archive of SID

که برخی برنامه‌ها برای پردازش فیلدهای طولانی‌تر از اندازه کلمه سیستم، آنها را در آرایه ذخیره می‌کنند. طبق مشاهدات ما یکی از الگوهای ذخیره در آرایه، مجموعه‌ای از دستورهای انتقالی متعلق به یک بلوک پایه از برنامه است که دنباله‌ای از بایت‌ها را در نشانی‌های حافظه متوالی ذخیره می‌کنند. در الگوریتم (۲) ابتدا این دستورها شناسایی می‌شوند و به‌صورت فهرستی در مجموعه α قرار می‌گیرند (خط ۱). در این الگوریتم بایت‌های عمل‌وند مبدأ دستورهای مجموعه α، به‌عنوان یک فیلد در نظر گرفته می‌شوند (خط ۵).

Algorithm 2	
	Input: $\theta, \mathcal{F}$
	Output: $\mathcal{F}$
	Begin
1:	$\alpha \leftarrow$ Set of instructions $\epsilon\ \theta$ that are used to move tainted bytes to arrays
2:	$s_o \leftarrow \phi$ //temporary list of offsets of tainted bytes that are used as source operand of an instruction
3:	**for each** list $lst_i\ \epsilon\ \alpha$
4:	$s_o \leftarrow$ offset of tainted bytes that are used as source operands in all instructions of lst_i
5:	**Create_Field** (s_o)
6:	**Update** $\mathcal{F}$
	End

در مرحله سوم، عمل‌وند دستورهای ارزیابی مورد بررسی قرار می‌گیرند. این دستورها شامل گستره وسیعی از دستورها هستند که در آنها مقدار یک پرچم برای مقایسه‌شدن در یک انشعاب شرطی تنظیم می‌شود. برای نمونه در دستور cmp مقدار پرچم صفر تنظیم می‌شود. اگرچه طبق استدلال ارائه‌شده در Icefex، این دستورها مدرک خوبی برای شناسایی محدوده فیلدها هستند؛ اما به‌دلایلی از جمله امکان مقایسه بخشی از فیلد و نیز وجود شروط ترکیبی ما این دستورها را در مرحله سوم مورد بررسی قرار می‌دهیم. در الگوریتم (۳) ابتدا مجموعه دستورهای مورد استفاده برای ارزیابی در مجموعه φ قرار می‌گیرند (خط ۱). با توجه به این که در دستورهای ارزیابی ممکن است، دو فیلد مختلف به‌عنوان عمل‌وندهای مبدأ و مقصد دستور قرار بگیرند و با هم مقایسه شوند، در این الگوریتم آفست بایت‌های عمل‌وند مبدأ و مقصد در مجموعه‌های d_o و s_o قرار داده می‌شوند (خط ۶ و ۷). با توجه به این که ممکن است، عمل‌وند دستورها با هم تداخل داشته باشند، ابتدا از هر دنباله بایتی که به‌عنوان عمل‌وند مبدأ دستور قرار گرفته است با استفاده از تابع create_fragment یک ساختار قطعه ایجاد می‌کنیم (خط ۸). ساختار قطعه یک ساختار فیلد موقتی است. درصورتی‌که بایت‌های عمل‌وند مبدأ و مقصد یکسان نباشند، با استفاده از بایت‌های عمل‌وند مقصد نیز قطعه‌هایی ایجاد می‌شود (خط ۱۰). وقتی بررسی عملوند همه دستورهای مجموعه φ به پایان رسید، با استفاده از تابع map_to_field قطعه‌ها را به فیلدها نگاشت می‌کنیم طوری که فیلدهای ایجادشده با هم تداخل نداشته باشند. برای نگاشت قطعه‌هایی که با هم تداخل دارند به فیلدها، قطعه‌ای با تعداد تکرار بیش‌تر، ابتدا به فیلد تبدیل می‌شود و قسمت باقی‌مانده قطعه دیگر، به فیلدی دیگر تبدیل شود. برای مثال درصورتی‌که نتیجه بررسی عمل‌وندهای مجموعه‌ای از دستورها دو قطعه باشد که یکی از آنها شامل دنباله بایت‌های (۱،۲،۳،۴) و تعداد تکرار ۵ باشد و قطعه دیگر شامل دنباله بایت‌های (۱،۲) و تعداد تکرار ۱۰ باشد فیلدهای حاصل عبارتند از یک فیلد با نقطه شروع ۱و نقطه پایان ۲ و یک فیلد با نقطه شروع ۳ و نقطه پایان ۴.

Algorithm 3	
	Input: $\theta, \mathcal{F}$
	Output: $\mathcal{F}$
1:	**Begin**
2:	$f \leftarrow \phi$ //temporary set of fragments
3:	$d_o \leftarrow \phi$
4:	$s_o \leftarrow \phi$
5:	$\varphi \leftarrow$ Set of evaluation instructions $\epsilon\ \theta$, those evaluate tainted bytes.
6:	**for each** instruction $ins_i\ \epsilon\ \varphi$
7:	$s_o \leftarrow$ offset of tainted bytes that are used as source operand of ins_i
8:	$d_o \leftarrow$ offset of tainted bytes that are used as destination operand of ins_i
9:	$f \leftarrow f \cup$ **create_fragment** (s_o)
10:	**If** $s_o \neq d_o$
11:	$f \leftarrow f \cup$ **create_fragment** (d_o)
12:	**map_to_field** (f)
	Update $\mathcal{F}$
	End

در مرحله چهارم عمل‌وند دستورهای مجموعه محاسباتی و منطقی مورد بررسی قرار می‌گیرند. این دستورها در مجموعه Δ قرار می‌گیرند. با توجه به این که ممکن است دستورهای مجموعه‌های φ و $\mathcal{C}$ نیز در این مجموعه قرار بگیرند، آنها را از مجموعه اصلی کم می‌کنیم (خط ۱). در اینجا نیز مشابه الگوریتم (۳) از ساخت قطعه‌ها و نگاشت آنها به فیلد برای شناسایی فیلدها استفاده می‌شود.

www.SID.ir

Archive of SID

Algorithm 4

	Input: $\theta, \mathcal{F}$
	Output: $\mathcal{F}$
	Begin
1:	$\Delta \leftarrow$ Set of logical and arithmetic instructions $\epsilon\ \theta \setminus (\varphi \cup \mathcal{C})$, that process tainted bytes.
2:	**for each** instruction ins$_i$ $\epsilon\ \Delta$
3:	$s_o \leftarrow$ offset of tainted bytes that are used as source operand of ins$_i$
4:	$d_o \leftarrow$ offset of tainted bytes that are used as destination operand of ins$_i$
5:	$f \leftarrow f \cup$ **create_fragment** (s_o)
6:	**if** $s_o \neq d_o$
7:	$f \leftarrow f \cup$ **create_fragment** (d_o)
8:	**map_to_field** (f)
9:	**Update** $\mathcal{F}$
	End

به‌دلیل آن که ممکن است، دستورهای انتقالی بدون توجه به محدوده فیلدها آنها را از مکانی به مکان دیگر انتقال دهند، در مرحله آخر در الگوریتم ۵ آنها را بررسی و مشابه الگوریتم‌های ۲ و ۳ از ساختن قطعه‌ها و نگاشت آنها به فیلد برای ساخت فیلد استفاده می‌کنیم. دستورهایی که در این الگوریتم پردازش می‌شوند، دستورهایی هستند که برای انتقال بایت‌های آلوده‌شده از رجیستر یا مکانی از حافظه به رجیستر یا مکان حافظه دیگری استفاده می‌شوند و عضو مجوعه α نیستند.

Algorithm 5

	Input: $\theta, \mathcal{F}$
	Output: $\mathcal{F}$
	Begin
1:	$\Psi \leftarrow$ Set of movement instructions $\epsilon\ \theta \setminus \alpha$, that transfer tainted bytes.
2:	**for each** instruction ins$_i$ $\epsilon\ \Psi$
3:	$s_o \leftarrow$ offset of tainted bytes that
4:	are used as source operand of ins$_i$
5:	$f \leftarrow f \cup$ **create_fragment** (s_o)
6:	**if** $s_o \neq d_o$
7:	$f \leftarrow f \cup$ **create_fragment** (d_o)
	$\mathcal{F} \leftarrow \mathcal{F} \cup$ **map_to_field** (f)
	End

برای فهم بیش‌تر الگوریتم‌ها، اجرای آنها را با درنظرگرفتن نمونه‌هایی بررسی می‌کنیم. فرض کنید ردّ اجرایی حاصل از پردازش پیام پروتکلی شامل ۲۳ بایت را در اختیار داریم و مجموعه $\mathcal{F} = \{[2,3]\}$ به‌عنوان ورودی داده شده است. در الگوریتم (۱) دستورهای مورد استفاده برای تبدیل نوع داده‌ای بایت به سایر انواع داده‌ای را در ردّ اجرایی جستجو می‌کنیم. مجموعه $\mathcal{C}$ به‌صورت زیر به‌دست می‌آید:

```
C: { [shl S0x8(T0),%ebx(T1,0)
 add %ebx(T1,0),%eax(T1,1)] }
```

آفست بایت‌هایی که به‌عنوان عمل‌وند مقصد دستورها استفاده شده‌اند یعنی صفر و یک به‌ترتیب در فهرست d_o وارد می‌شوند و درنهایت فهرست d_o به‌عنوان ورودی تابع create_field داده می‌شود. درنتیجه مجموعه $\mathcal{F}$ به‌صورت $\mathcal{F} = \{[2,3],[0,1]\}$ به‌روز می‌شود. در الگوریتم (۲) دستورهای مورد استفاده برای انتقال بایت‌های آلوده‌شده به آرایه در ردّ اجرایی جستجو می‌شوند. مجموعه α به‌صورت زیر به‌دست می‌آید:

```
α :{ [ mov %edx,(%eax):
   R@edx (T1,7,8,9,10),  M@0x12d964 (T0);
   // R shows Register name and M shows Memory
   address.
   mov %edx,0x4(%eax):
   R@eax(T1,11,12,13,14), M@0x12d968 (T0);
   mov %edx,0x8(%eax):
   R@eax (T1,15,16,17,18), M@0x12d96c (T0);
   mov %ecx,0xc(%eax):
   R@ecx(T1,19,20,21,22), M@0x12d970 (T0);]}
```

در این مجموعه تنها یک فهرست از دستورها وجود دارد که در آن بایت‌های با آفست ۷ تا ۲۲ از پیام ورودی در نشانی‌های حافظه 0x12d964 تا 0x12d974 نوشته می‌شوند و بنابراین برای پردازش این مجموعه دستورها، حلقه موجود در الگوریتم فقط یک مرتبه اجرا می‌شود. با استخراج عملوند مبدأ دستورها، فهرست s_o به‌صورت [7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22] حاصل می‌شود. بنابراین مجموعه $\mathcal{F}$ به‌صورت $\mathcal{F} = \{[2,3], [0,1], [7,22]\}$ به‌روز می‌شود. در الگوریتم (۳) دستورهای مورد استفاده برای ارزیابی را جستجو می‌کنیم. مجموعه φ به‌صورت زیر به‌دست می‌آید:

```
φ :{ cmp  S0x3,%eax: I@0x3(T0),R@eax(T1,6);
    //I shows immediate value}
```

این مجموعه شامل یک دستور است که در آن عملوند مبدأ تهی و عملوند مقصد معادل بایت آلوده‌شده با آفست ۶ از پیام ورودی است. بنابراین s_o=∅ و d_o=[6] و $f = \{[6,6]\}$ می‌شوند. پس از نگاشت قطعه به فیلد مجموعه $\mathcal{F}$ به‌صورت $\mathcal{F} = \{[2,3],[0,1],[7,22],[6,6]\}$ به‌روز می‌شود. در الگوریتم (۴) دستورهای منطقی و محاسباتی در ردّ اجرایی جستجو می‌شوند. مجموعه $\Delta = \emptyset$ در نتیجه حاصل می‌شود. این مجموعه تهی است و دستوری برای پردازش وجود ندارد. بنابراین به الگوریتم (۵) می‌رویم. در این الگوریتم دستورهای

www.SID.ir

Archive of SID

مورد استفاده برای انتقال جستجو می‌شوند. مجموعه Ψ به‌صورت زیر به‌دست می‌آید:

```
:{ mov %eax, 0x59f264: Ψ
  R@cax(T1,4,5),M@0x59f264(T0);
   mov %eax,%ebx:
  R@eax(T1,4), R@ebx(T0);
   mov %ecx,%ebx
  R@ecx(T1,4,5), R@ebx(T0);}
```

در نخستین مرتبه از اجرای حلقه s_o=[4,5] و d_o=∅ می‌شود و مجموعه قطعه‌ها f = {[4,5]} می‌شود. در مرتبه دوم اجرای حلقه s_o= [4] و d_o=∅ و مجموعه قطعه‌ها f = {[4,5],[4,4]} می‌شود. در مرتبه سوم اجرای حلقه s_o=[4,5] و d_o=∅ و مجموعه قطعه‌ها f ={[4,5],[4,4],[4,5]} می‌شود. پس از اتمام اجرای دستورهای حلقه باید قطعه‌ها به فیلدها نگاشت شوند تا فیلدهای حاصل تداخل نداشته باشند. با توجه به این که قطعه [4,5] دو مرتبه و قطعه [4,4] یک مرتبه تکرار شده است، نتیجه تابع نگاشت قطعه به فیلد، فیلد [4,5] را به‌عنوان خروجی اعلام می‌کند و مجموعه $\mathcal{F}$ به‌صورت $\mathcal{F}$ = {[2,3],[0,1],[7,22],[6,6],[4,5]} به‌روز می‌شود و روش تشخیص محدوده فیلدها با موفقیت پایان می‌یابد.

۴-۴- تشخیص فیلد نوع

یکی از پراستفاده‌ترین فیلدها در پروتکل‌ها، فیلد نوع است. فیلد نوع برای تشخیص کلاس‌های مختلف یک شیء استفاده می‌شود. کدهای عملیاتی[1]، مثال‌های معمولی از فیلد نوع هستند. شناسایی چنین فیلدهایی در برخی از کاربردهای امنیتی مورد استفاده قرار می‌گیرد. برای نمونه در سامانه‌های تشخیص نفوذ مبتنی بر ناهنجاری، با اطلاع‌داشتن از کد عملیات‌های پروتکل، قرارگیری مقادیری به جز کد عملیات‌ها در مکان کد عملیاتی به‌عنوان رفتاری غیر نرمال و ناهنجار در نظر گرفته می‌شود [39].

شیوه پیشنهادی ما برای تشخیص فیلد نوع بر پایه این استدلال است که مقادیر مجاز فیلد نوع، مجموعه معین و محدودی از داده‌ها هستند که برنامه آنها را از قبل می‌داند. برنامه نیاز به دانستن مقدار این فیلدها برای تصمیم‌گیری در مورد شیوه پردازش پیام دارد؛ این کار با مقایسه بایت‌های ورودی و مقادیر ثابت انجام می‌شود؛ بنابراین ما فیلد نوع را با جستجوی مقایسه بایت‌های آلوده‌شده و مقادیر ثابت شناسایی می‌کنیم. چون ممکن است در هر ردّ اجرایی تعداد محدودی از این مقایسه‌ها وجود داشته باشد، برای افزایش دقت و شناسایی کامل‌تر و همچنین اطلاع از سایر مقادیر مجاز فیلد نوع، تحلیل پویای کد دودویی را با تحلیل ایستا ترکیب می‌کنیم. الگوریتم (۶)، شبه‌کد شیوه شناسایی فیلد نوع را نشان می‌دهد. این الگوریتم فیلدی را به‌عنوان ورودی می‌گیرد و تشخیص می‌دهد که فیلد نوع است یا خیر و درصورتی‌که فیلد نوع باشد، مجموعه‌ای از برخی مقادیر مجاز این فیلد را نیز استخراج می‌کند. دانستن این مجموعه مقادیر به ما کمک می‌کند تا درصورتی‌که در ترافیک شبکه، پیام‌هایی از یک پروتکل ناشناخته مشاهده شد و این مجموعه مقادیر در مکان فیلد نوع ظاهر می‌شدند، تشخیص دهیم ترافیک مبادله‌شده متعلق به پروتکل ناشناخته یادشده است. همچنین فازرها برای تولید ورودی‌های بدخواه از این اطلاعات استفاده می‌کنند.

Algorithm 6- Type Field Detection

```
     Input: θ,d , 𝓕
     // θ: Set of all instructions in execution trace.
     // d: Set of all instructions of disassembled
     code.
     // 𝓕: a field of message
1:   Output: Result, e
     // Result identifies that the field is type or not.
2:   // e: set of enumeration values of type field
3:   begin
4:   φ ← set of all evaluation instructions ∈ θ, that
5:   evaluate 𝓕or part of it.
     τ ← temporary set of evaluation instructions
6:   func ← temporary set of instructions
     for each instruction insᵢ ∈ φ
7:       if insᵢ compares the equality or inequality
8:   of          𝓕 with a constant value K:
                 e ← e ∪ { K }
9:           //add constant value to e
10:          func ← find_function_ins (𝔇, insᵢ)
             τ ← set of evaluation instructions ∈
     func that evaluate 𝓕or part of it.
11:              for each instruction instₖ ∈ τ
12:                      if instₖ compares the equality
13:  or inequality of 𝓕 with a constant value K′ and
14:  K ≠ K′ :
15:                          Result ← yes
16:                          e ← e ∪ {k′ }
     if Result is yes:
      return Result,e
     else
      return Result ← no
     end
```

درکل شیوه کار این الگوریتم به این صورت است که ابتدا دستورهایی را که برای ارزیابی فیلد ورودی یا بخشی از آن در ردّ اجرایی وجود دارند، شناسایی می‌کنیم و آنها را در

[1] Operational Codes (OPcode)

www.SID.ir

Archive of SID

مجموعه φ قرار می‌دهیم. در این دستورها مقدار یک پرچم تنظیم شده و سپس در یک انشعاب شرطی مقایسه می‌شود. در این مجموعه هر دستور ارزیابی، شامل دستور تنظیم پرچم و دستور انشعاب شرطی مرتبط با آن است. برای نمونه، [cmp eax, ebx; je 0xa0db78c7] یکی از اعضای مجموعه دستورهای ارزیابی است. در مرحله بعد در مجموعه دستورهای ارزیابی دستوری را می‌یابیم که در آن برابربودن یا برابرنبودن فیلد یا بخشی از آن با مقداری ثابت مقایسه می‌شود (خط ۵ الگوریتم). اگر دستوری با این مشخصات در ردّ اجرایی وجود داشت، دستورهای تابعی که این دستور متعلق به آن است را می‌یابیم. برای این کار از تابع *find_function_ins* استفاده می‌شود. اگر در این تابع دستورهای ارزیابی دیگری وجود داشتند که برابربودن یا نبودن مقدار فیلد یادشده را با مقدار ثابت دیگری بررسی کنند (خط ۱۰ الگوریتم)، این فیلد را به‌عنوان فیلد نوع معرفی می‌کنیم و تمامی مقادیر ثابتی که فیلد نوع با آنها مقایسه می‌شود را در یک مجموعه قرار می‌دهیم (خط ۱۲ الگوریتم) و به‌عنوان خروجی اعلام می‌کنیم. به این ترتیب به‌عنوان نمونه با استفاده از این روش در پیام درخواست پروتکل HTTP، مقدار GET به‌عنوان فیلد نوع شناسایی شده است. سایر مقادیر معتبری که برای این فیلد با استفاده از این روش به‌دست می‌آید، عبارتند از: Post، Put، Checkin، CheckOut، Propfind، Delete، Report، Update، Unlock و غیره.

برای فهم بهتر طرز کار الگوریتم، اجرای این الگوریتم را با درنظرگرفتن یک نمونه بررسی می‌کنیم. فرض کنید ردّ اجرایی حاصل از پردازش پیام پروتکلی را در اختیار داریم و می‌خواهیم تعیین کنیم آیا فیلد [2,3] یک فیلد نوع است یا خیر. برای این کار دستورهای ارزیابی فیلد را در ردّ اجرایی جستجو می‌کنیم. مجموعه φ به‌صورت زیر به‌دست می‌آید.

```
{[589269: cmp  $0x5, %cax: I@(T0), R@cax(T1,2,3);
 58926b: jg     0x005891d8]
[5891c8: cmp S0x3, 0x59ec63:I@(T0),
M@0x0059ec63 (T1,2,3) ;
5891ca: je     0x005891d8] }
```

در این مجموعه هر دستور با نشانی آن و اطلاعات آلودگی پس از پایان دستور مشخص شده است. (T1,2,3) نشان‌دهندهٔ آن است که در این مکان حافظه، بایت‌های آلودشده ۲ و ۳ قرار دارند. در نخستین دستور ارزیابی این مجموعه، مقدار ثابت 0x5 با فیلد [2,3] در رابطه بزرگ‌تری مقایسه می‌شوند. بنابراین دومین عضو مجموعه را مورد بررسی قرار می‌دهیم. در دومین دستور ارزیابی مقدار ثابت 0x3 با فیلد [2,3] به‌صورت برابری مقایسه می‌شود، بنابراین مقدار 0x3 به مجموعه e افزوده می‌شود؛ سپس در مجموعه دستورهای دیس‌اسمبل‌شده کد دودویی برنامه، تابعی را می‌یابیم که این دستور متعلق به آن است و دستورهای این تابع را در مجموعه *func* قرار می‌دهیم. پس از آن دستورهای مورد استفاده برای ارزیابی فیلد [2,3] را که در مجموعه *func* قرار دارند، می‌یابیم و آنها را در مجموعه τ قرار می‌دهیم. مجموعه τ به‌صورت زیر به‌دست می‌آید:

```
{ [589265: cmp $0x16, 0x59ec63;589267: je
x0058926d]
[5892cb:cmp $0x17,0x59cc63; 5892f2: jnc
x00589481]}
```

در همه این دستورها نشانی حافظه‌ای که فیلد [2,3] در آن قرار دارد با مقداری ثابت به‌صورت برابری یا نابرابری مقایسه شده است، بنابراین همه این مقادیر ثابت در مجموعه e قرار می‌گیرند و $e = \{0x3, 0x16, 0x17\}$ می‌شود. همچنین مقدار متغیر *Result*، yes می‌شود. پس فیلد [2,3] یک فیلد نوع است و مجموعه مقادیر مجاز آن عبارتند از: 0x3، 0x16 و 0x17.

۵-۴- شناسایی معنای فیلدها

یکی از ویژگی‌های مهم هر فیلد، معنای آن است. معنای فیلد نشان‌دهندهٔ نوع اطلاعاتی است که فیلد در خود نگه می‌دارد. مثال‌های معمولی از معنای فیلد عبارتند از طول، نشانی IP، شماره درگاه، نام فایل و غیره. دانستن معنای فیلد برای فهمیدن هدف پیام و پروتکل ضروری است؛ به‌علاوه معنای فیلدها در بسیاری از کاربردهای امنیتی مورد استفاده قرار می‌گیرد؛ برای مثال برای کاهش هشدارهای غلط در امضاهای تولیدشده برای سامانه‌های تشخیص نفوذ، برای تولید امضاها از فهم معانی پروتکل‌های لایه کاربرد استفاده می‌شود [40].

در میان روش‌های ارائه‌شده تاکنون، Dispatcher برای شناسایی معانی ازپیش تعیین‌نشده فیلدها از این مشاهده استفاده می‌کند که بسیاری از توابع و دستورهای استفاده‌شده در برنامه حاوی اطلاعات معنایی هستند که می‌توانند برای شناسایی معانی فیلدها مورد استفاده قرار بگیرند. وقتی آرگومان ورودی چنین توابع یا دستورهایی از فیلدی از پیام به‌دست آید، معنای آن فیلد می‌تواند با استفاده از اطلاعات پیش‌نمونه آن توابع شناسایی شود. اگرچه با استفاده از این روش، معانی برخی فیلدها قابل استخراج‌شدن است، اما این روش محدودیت‌هایی نیز دارد، از جمله این که بیش‌تر توابع

www.SID.ir

Archive of SID

مورد استفاده برای پردازش فیلدها توسط کاربر تعریف شده‌اند و اطلاعات پیش‌نمونهٔ آنها در دسترس نیست؛ همچنین بسیاری از فیلدها به‌عنوان آرگومان هیچ تابعی استفاده نمی‌شوند؛ بنابراین به روش‌هایی دیگر برای شناسایی معانی فیلدهایی که ناشناخته باقی می‌مانند نیاز داریم. Netzob نیز برای شناسایی معانی فیلدها، آنها را در اطلاعات زمینه‌ای و محیطی مانند اطلاعات اتصال به شبکه، فایل‌های دسترسی‌شده و غیره جستجو می‌کند. معانی به‌دست‌آمده با استفاده از این روش بسیار محدود هستند؛ بنابراین به روش‌های دیگری برای شناسایی کامل‌تر معانی نیاز است.

در این مقاله، روش جدیدی برای استخراج معانی گسترده‌ای از فیلدهای پیام ارائه می‌دهیم. روش پیشنهادی ما بر اساس این استدلال است که یک برنامه کاربردی نیاز به برقراری ارتباط با کاربر و نمایش اطلاعات قابل فهم برای او دارد؛ اگر پردازش فیلدی منجر به نمایش اطلاعات برای کاربر شود، می‌توانیم با استفاده از اطلاعات نمایش‌داده‌شده برای کاربر که توسط انسان قابل فهم است، معنای فیلد را شناسایی کنیم. برای مثال اگر فیلدی مقدار نامعتبری داشته باشد و پس از پردازش آن، پیام Bad protocol identifier برای کاربر نمایش داده شود، می‌توان فهمید که فیلد مورد نظر اطلاعاتی در مورد protocol identifier در خود دارد.

اطلاعات نمایش‌داده‌شده برای کاربر، یا به‌طور مستقیم حاوی معانی قابل فهم برای کاربر هستند، مثل رشته[1]‌های موجود در برنامه، یا به واسطه اطلاعات مکانی که در آن قرار می‌گیرند برای کاربر قابل فهم می‌شوند، مثل مقادیری که در ستونی از یک پایگاه داده قرار می‌گیرند. در حالت دوم، باید توابعی که برای ارتباط با منابع برنامه استفاده شده‌اند (مانند تابع sendmessage) را شناسایی کرد؛ سپس با توجه به اطلاعات موجود در پیش‌نمونه این توابع و تطبیق آنها با خروجی ابزار تحلیل منابع برنامه، قسمتی از منبع که قرار است، داده آن تغییر یابد، شناسایی کرد. برای مثال، باید شناسه شیئی[2] از دیالوگ[3] واسط گرافیکی را که قرار است، مقدار یکی از بخش‌های آن تغییر یابد با استفاده از اطلاعات موجود در پیش‌نمونه تابع sendmessage به‌دست آورد و از تطبیق آن با اطلاعات ابزار تحلیل منابع، بخش مورد نظر را شناسایی کرد.

برای دلالت بر این که اطلاعات برای کاربر نمایش داده می‌شوند اگر و فقط اگر فیلد پردازش شود، از رابطه تسلط[4] موجود میان رأس‌های گراف جریان اجرا استفاده می‌کنیم. رابطه تسلط، رابطه‌ای است که میان رأس‌های گراف به‌صورت زیر برقرار است:

تعریف (رابطه تسلط): رأس x بر رأس y تسلط دارد اگر تمام مسیرها از نقطه آغازین گراف تا رأس y، به‌حتم شامل رأس x باشند [41].

طبق تعریف رابطه تسلط، اگر ارزیابی فیلدی در رأس x و اطلاعات نمایش‌داده‌شده برای کاربر در رأس y قرار داشته باشد و x بر y تسلط داشته باشد، یعنی در هر مسیر از ابتدای گراف جریان اجرا تا رأس y، رأس x به‌حتم حضور دارد و درصورتی‌که رأس x در مسیری نباشد، آن مسیر به رأس y نخواهد رسید؛ پس درصورتی‌که فیلد مورد نظر ارزیابی نشود، اطلاعات نیز برای کاربر نمایش داده نمی‌شود. به این ترتیب یک رابطه معنایی میان فیلد ارزیابی‌شده و اطلاعات نمایش‌داده‌شده برای کاربر (برای سادگی رشته‌های مشاهده‌شده را در نظر می‌گیریم.) وجود دارد که آن را به‌صورت زیر تعریف می‌کنیم:

تعریف (رابطه معنایی): اگر فیلد F در رأس x با استفاده از شرطی ساده ارزیابی شود و رشته S در دستوری از رأس z وجود داشته باشد F و S ارتباط معنایی دارند و با رابطه $F \bowtie S$ نمایش داده می‌شود، اگر و فقط اگر (۱) x بر z تسلط داشته باشد و (۲) رأس y ای وجود نداشته باشد که فیلد F′ را ارزیابی کند و x بر y و y بر z تسلط داشته باشد.

شرط دوم در این رابطه به این دلیل بررسی می‌شود که ممکن است، در مسیر رأس x تا رأس z، رأس دیگری وجود داشته باشد که فیلد دیگری از پیام را ارزیابی می‌کند که مشاهدهٔ رشتهٔ نتیجه ارزیابی آن است. بنابراین درصورتی ارتباط معنایی برقرار است که چنین رأسی وجود نداشته باشد.

به‌دلیل آن که ممکن است، برای ارزیابی فیلدی از شروط ترکیبی استفاده شود، رابطه معنایی ترکیبی را به‌صورت زیر تعریف می‌کنیم:

تعریف (رابطه معنایی ترکیبی): اگر فیلد F در رأس x ارزیابی شود و برای ارزیابی آن از شرطی ترکیبی استفاده شده باشد و رشته S در دستوری از رأس z وجود داشته باشد، S با تمام فیلدهایی که در شرط ترکیبی ارزیابی می‌شوند، ارتباط معنایی دارد، اگر و فقط اگر F و S رابطه معنایی داشته باشد.

---

[1] Strings

[2] Object

[3] Dialog

[4] Domination

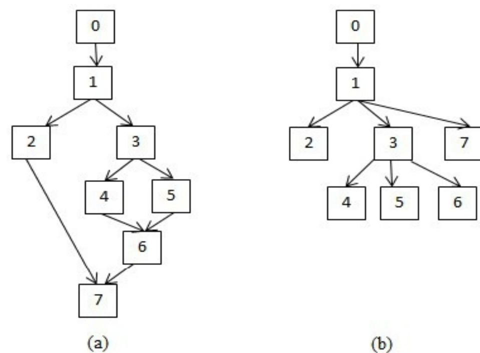
www.SID.ir

Archive of SID

برای پیاده‌سازی روش پیشنهادی، نیاز به استخراج روابط تسلط موجود در گراف جریان اجرا داریم. این روابط، در ساختاری درختی که درخت تسلط نامیده می‌شود، نمایش داده می‌شوند. در درخت تسلط، فرزند هر رأس، رأسی است که بر آن رابطه تسلط بدون واسطه[1] دارد. رابطه تسط بدون واسطه به‌صورت زیر تعریف می‌شود:

تعریف (رابطه تسلط بدون واسطه): رأس x بر رأس z به‌صورت بدون واسطه تسلط دارد اگر (۱) رأس x بر رأس z تسلط داشته باشد و $x \neq z$ (۲) اگر رأس x بر رأس y تسلط دارد، رأس y بر رأس z تسلط نداشته باشد و $y \neq z$ [41].

برای نمونه، درصورتی‌که گراف جریان اجرا به‌صورت شکل (۳)(a) باشد، درخت تسلط آن به‌صورت شکل (۳)(b) خواهد بود.



(شکل-۳): (a) نمونه گراف جریان اجرا (b) درخت تسلط معادل آن.

(Figure-3): (a) The control flow graph and (b) the corresponding dominator tree.

به این ترتیب با دراختیارداشتن درخت تسلط درصورتی‌که در فرزندان رأسی که دستور ارزیابی فیلد در آن قرار دارد، رشته یا تابع ارسال اطلاعات به منابع برنامه وجود داشته باشد، می‌توانیم معنای فیلد را به‌دست می‌آوریم. الگوریتم (۷) شیوه پیشنهادی تشخیص معانی بر این اساس را نشان می‌دهد. ورودی‌های این الگوریتم عبارتند از: دستورهای ردّ اجرایی، درخت تسلط گراف جریان اجرای کد دودویی برنامه پیاده‌ساز پروتکل و مجموعه دستورهای موجود در ردّ اجرایی که برای ارزیابی فیلدها استفاده شده‌اند و خروجی الگوریتم مجموعه‌ای از رابطه‌های معنایی میان فیلدها و رشته‌ها است.

در ابتدا از تابع find_node برای شناسایی گره‌ای که دستور ارزیابی در آن قرار گرفته است، استفاده می‌کنیم (خط ۵). این تابع، نشانی یک دستور و یک ساختار درخت را به‌عنوان ورودی می‌گیرد و گره‌ای از درخت را که نشانی مورد نظر در آن قرار دارد به‌عنوان خروجی اعلام می‌کند؛ سپس در فرزندان این گره از درخت تسلط، رشته‌ها را به روش پیمایش عمق نخست جستجو می‌کنیم (خط ۶). جستجوی عمق نخست در درخت تسلط این امکان را به ما می‌دهد که از مقدار فیلدی که سبب مشاهده رشته بوده است، اطلاع پیدا کنیم. از تابع depth_first_search_strings برای جستجوی رشته‌ها به روش عمق نخست استفاده می‌شود. خروجی این تابع در $\mathcal{S}$ قرار داده می‌شود که مجموعه‌ای از دوتایی‌های <string , instruction> است که در آن instruction، نشانی دستوری را که رشته در آن قرار دارد، نشان می‌دهد. اگر $\mathcal{S}_k$، k اُمین عضو مجموعه را نشان دهد از نماد $\mathcal{S}_k$.string برای اشاره به رشته این عضو و از نماد $\mathcal{S}_k$.instruction برای اشاره به دستوری که رشته در آن قرار دارد استفاده می‌کنیم.

Algorithm 7- Semantic Field Inference	

1:
2:
 	
3:
4:
5:
6:
 	
7:
8:
9:
10:
11:
12:
13:
14:
15:
	Input: $\theta, \mathfrak{D}, \varphi$
	//θ: Set of all instructions in execution trace
	//$\mathfrak{D}$: Dominator Tree that is produced from control flow graph of the program.
	//φ: Set of all evaluation instructions $\in \theta$, that evaluate tainted bytes.
	output: ∂
	//∂: Set of semantic relations between fields and strings
	begin
	// x, y: nodes in $\mathfrak{D}$
	for each instruction $ins_i \in \varphi$
	$\mathcal{S} \leftarrow \emptyset$: set of 2-tuples <string, instruction>
	$f \leftarrow \emptyset$: temporary set of fields
	$\mathcal{F} \leftarrow$ the field that ins_i evaluates it.
	$x \leftarrow$ **find_node** ($ins_i, \mathfrak{D}$)
	$\mathcal{S} \leftarrow$ **depth_first_search_strings**($\mathfrak{D}, x$)
	//search by starting from node x in dominator tree
	for each string $\mathcal{S}_k$.string $\in \mathcal{S}$
	$y \leftarrow$ **find_node**($\mathcal{S}_k$.instruction, $\mathfrak{D}$)
	if semantic_relation (x, y) = true
	$\partial \leftarrow \partial \cup \{\mathcal{S}_k.\text{string} \bowtie \mathcal{F}\}$
	$f \leftarrow$ **compound_condition**(ins_i)
	if $f \neq \emptyset$
	for each field $f_i \in f$
	$\partial \leftarrow \partial \cup \{\text{string}_i \bowtie F\}$
	return ∂
	end

در مرحله بعد باید بررسی شود که فیلد با کدام یک از رشته‌های درون مجموعه $\mathcal{S}$ ارتباط معنایی دارد. برای این که ارتباط معنایی میان فیلد و رشته برقرار باشد، در مسیر میان

[1] Immediate

www.SID.ir

Archive of SID

گره ارزیابی فیلد تا گره شامل رشته نباید گره دیگری وجود داشته باشد که فیلد دیگری از پیام را ارزیابی کند. تابع semantic_relation وظیفه بررسی این مورد را بر عهده دارد (خط ۹)، به این صورت که در گره‌های میان ارزیابی فیلد تا گره‌ای که رشته در آن قرار دارد، دستورهای ارزیابی را در یک مجموعه قرار می‌دهد؛ سپس نشانی این دستورها را در ردّ اجرایی جستجو می‌کند، اگر موردی شناسایی شد که در آن فیلدی از پیام ارزیابی می‌شود semantic_relation مقدار false را باز می‌گرداند؛ اما درصورتی‌که چنین دستوری در ردّ اجرایی شناسایی نشود، مقدار true بازگردانده می‌شود، یعنی رابطه معنایی برقرار است و بنابراین این رابطه معنایی به مجموعه ∂ افزوده می‌شود (خط ۱۰). همچنین باید بررسی کنیم رابطه معنایی، ترکیبی نیز هست یا خیر (خط ۱۱). به این منظور باید نوع شرط مورد استفاده برای ارزیابی فیلد مشخص شود. تابع compound_condition وظیفه شناسایی شرطی یا ساده‌بودن نوع شرط را بر عهده دارد و درصورتی‌که شرط، ترکیبی باشد، سایر فیلدهایی که در این شرط ترکیبی ارزیابی می‌شوند، نیز با رشته مورد نظر ارتباط معنایی دارند و این ارتباط‌های معنایی نیز به مجموعه ∂ افزوده می‌شوند (خط ۱۴). در نهایت پس از اتمام اجرای حلقه پردازش دستورهای مجموعه ارزیابی، مجموعه ∂ به‌عنوان خروجی الگوریتم برگردانده می‌شود.

برای فهم بیش‌تر اجرای الگوریتم را همراه با مثالی ساده توضیح می‌دهیم. فرض کنید یکی از دستورهای عضو مجموعه φ به‌صورت زیر است:

```
588d9d: cmp  $0x0,0x49cc5d:            I@0x0(T0),
M@0x49cc5d(T1,2,3);
```

گره‌ای از درخت تسلط (گره x) را که این دستور در آن قرار دارد، می‌یابیم (خط ۵). در فرزندان این گره به‌صورت عمق نخست رشته‌ها را جستجو می‌کنیم و آنها را همراه با نشانی دستورالعمل‌شان در مجموعه S قرار می‌دهیم (خط ۶). فرض کنید، مجموعه S به‌صورت زیر حاصل شده است:

```
{<"Bad Protocol Identifier", 57c4a7>, <" illegal
  function", 588dba>}
```

برای هر یک از اعضای این مجموعه باید بررسی شود آیا میان رشته‌های عضو و فیلد [2,3] ارتباط معنایی برقرار است یا نه. گره متناظر با رشته نخست مجموعه (گره y) را در تسلط جستجو می‌کنیم (خط ۸). در خط ۹ الگوریتم بررسی می‌شود آیا میان گره x و y ارتباط معنایی برقرار است یا خیر. با توجه به این که در مسیر میان گره x تا گره y درخت تسلط، دستور ارزیابی وجود ندارد، بنابراین رابطه معنایی برقرار است و مجموعه ∂ به‌صورت

{"Bad protocol identifier" ⋈ [2,3]}

می‌شود (خط ۱۰). با اجرای تابع compound_condition یک مجموعه تهی به‌دست می‌آید. درنتیجه دوباره به ابتدای حلقه (خط ۷) برمی‌گردیم. عضو دوم مجموعه S را در نظر گرفته و گره متناظر با آن (گره z) را در درخت تسلط جستجو می‌کنیم (خط ۸). با اجرای تابع semantic_relation، دستورهای ارزیابی موجود در مسیر میان گره x تا گره z درخت تسلط در یک مجموعه قرار داده می‌شود. در این مجموعه، دستوری با نشانی 57c45b قرار می‌گیرد. این نشانی را در ردّ اجرایی جستجو می‌کنیم. دستور زیر در نتیجه شناسایی می‌شود:

```
57e45b: test   $0x0, %eax: I@0x0 (T0), R@ (T1,
4,5);
```

همان‌طورکه مشخص است، این دستور، فیلد [4,5] را ارزیابی می‌کند، بنابراین خروجی تابع semantic_realtion، false خواهد بود و مجموعه ∂ تغییری نخواهد کرد. به‌دلیل این که اعضای مجموعه S به پایان رسیده است، الگوریتم پایان می‌یابد.

اگرچه با استفاده از این روش، معانی گسترده‌ای از پیام استخراج می‌شود، اما ممکن است، معنای برخی فیلدها ناشناخته باقی بماند، زیرا در پردازش برخی فیلدها ارتباطی با کاربر برقرار نمی‌شود؛ در این‌گونه موارد درصورتی‌که پیاده‌سازی‌های دیگری از پروتکل موجود باشد، آنها را بررسی می‌کنیم. درصورتی‌که باز هم معنای فیلد شناسایی نشد، ابتدا مشابه روش استفاده‌شده در Dispatcher، درصورتی‌که بایت آلوده‌شده‌ای به‌عنوان آرگومان تابعی شناخته‌شده استفاده شود با استفاده از اطلاعات موجود در پیش‌نمونه آن معنای فیلد را شناسایی می‌کنیم. برای فیلدهایی که معانی آنها هنوز ناشناخته باقی مانده است، روش استفاده‌شده در Netzob برای استخراج معانی را گسترش می‌دهیم و این فیلدها را در اطلاعات محیطی برنامه جستجو می‌کنیم. اطلاعات محیطی برنامه را اطلاعاتی تعریف می‌کنیم که برای ایجاد ارتباط و هماهنگی میان دو طرف درگیر در ارتباط مبادله می‌شوند و نیز اطلاعات مکان‌هایی که در زمان پردازش پیام مورد دسترسی قرار می‌گیرند. چهار دسته از این اطلاعات عبارتند از:

۱- اطلاعات مربوط به اتصال به شبکه مانند نشانی IP و شماره درگاه

www.SID.ir

Archive of SID

۲- اطلاعات نصب و پیکربندی برنامه مانند نسخه برنامه مورد استفاده و اطلاعات سیستم عامل مورد استفاده: برای مثال ممکن است، برنامه سمت مشتری[1] نسخه برنامه مورد استفاده خود را برای برنامه سمت خدمت‌کار[2] بفرستد تا در صورت وجود نسخهٔ به‌روزشده[3]، آن را دریافت کند.

۳- اطلاعات مورد نیاز برای احراز هویت کاربر مانند نام و شناسه کاربری

۴- اطلاعات فایل‌های دسترسی‌شده، کلیدهای تغییریافته رجیستری، قسمت‌های تغییریافته واسط گرافیکی و غیره.

۵- پیاده‌سازی و ارزیابی

در این بخش ابتدا جزئیات پیاده‌سازی سامانه و شیوه ارزیابی آن را شرح می‌دهیم؛ سپس جزئیات آزمایش‌های انجام‌شده و نتایج آنها ارائه می‌شود.

۵-۱- پیاده‌سازی

جزئیات پیاده‌سازی پنج ماژول مورد استفاده در سامانه پیشنهادی در زیر آمده است:

نظارت بر اجرا: برای نظارت بر اجرای برنامه در هنگام پردازش پیام ورودی شبکه از Temu [42] استفاده می‌کنیم که بر روی Qemu [46] (یک نماساز[4] کلی سامانه) پیاده‌سازی شده است. Temu برنامه را در سامانه نماسازی‌شده اجرا می‌کند و می‌تواند همه دستورهای پردازش‌شده در زمان اجرا را ضبط کند. همچنین می‌تواند داده‌های ورودی به برنامه از منابع مختلف (مانند ورودی‌های کارت شبکه و صفحه کلید) را به‌عنوان بایت‌های آلوده‌شده علامت بزند و انتشار آنها را در طول اجرای برنامه ردگیری[5] کند. خروجی این ابزار، یک ردّ اجرایی است که در آن دستورهای پردازش داده‌های آلوده‌شده، همراه با مقادیر عملوندها و اطلاعات آلودگی وجود دارد.

جمع‌آوری اطلاعات محیطی: برای جمع‌آوری بخشی از اطلاعات محیطی از جعبه شنی[6] Buster [20] استفاده می‌کنیم که ابزاری برای تحلیل رفتار برنامه‌ها است. این ابزار قابلیت مشاهدهٔ ارتباطات برقرارشده در شبکه و تغییرات ایجادشده در سامانه مانند تغییرات فایل‌ها و رجیستری را دارد.

دیس‌اسمبلر: برای دیس‌اسمبل‌کردن دودویی برنامه‌ها از دیس‌اسمبلر مشهور IDA Pro [45] استفاده کرده‌ایم.

تحلیل‌گر دیس‌اسمبلی: برای تحلیل کد دیس‌اسمبلی تولیدشده، از Zynamics Binnavi [46] استفاده کرده‌ایم. این ابزار می‌تواند حلقه‌های موجود در کد را استخراج کند و گراف جریان اجرا و درخت تسلط را تولید کند.

تحلیل‌گر منابع: برای مشاهده اطلاعات منابع مورد استفاده در برنامه از ابزار Resource tuner [47] که یک ابزار تحلیل ایستا برای مشاهده، ویرایش و استخراج داده از بخش منابع[7] یک فایل اجرایی است، استفاده کرده‌ایم.

۵-۲- شیوه ارزیابی

با توجه به این که پژوهش‌های پیشین در استخراج مشخصات پروتکل‌های دودویی-کدشده محدودیت دارند، برای نشان‌دادن کارایی سیستم پیشنهادی خود، شش پیام متعلق به چهار پروتکل دودویی-کدشده و شناخته‌شده DNS، eDonkey، Modbus و STUN را تحلیل کردیم. برنامه‌های پیاده‌ساز هر پروتکل در جدول (۱) نمایش داده شده‌اند؛ اندازه هر برنامه نیز در جدول (۱) نمایش داده شده تا درصورتی‌که چند نسخه از برنامه موجود است، فایل اجرایی اصلی مشخص شود. برای ارزیابی سامانه پیشنهادی خود روش‌های پیشنهادی در این مقاله و فنون استفاده‌شده در Dispatcher (یکی از پژوهش‌های اخیر که روشی را برای استخراج معانی نیز ارائه کرده است) به‌طور جداگانه برای استخراج فرمت این پیام‌ها استفاده کردیم و نتایج حاصل را با نتایج خروجی از ابزار Wireshark [48]، به‌عنوان مرجع مقایسه کرده‌ایم. این نتایج در زیربخش ۵-۳ با جزئیات نشان داده خواهند شد.

(جدول-۱): برنامه‌های مختلف مورد استفاده برای ارزیابی.
(Table-1): Different program binaries used in our evaluation.

Protocol	Server	Binary size (KB)	Client	Binary size (KB)
DNS	Bind 9.9.8	396	nslookup	107
eDonkey	eserver 17.13	564	emule 0.48a	5.184
Modbus	Ananas 1.8.1.0	2.356	Ananas 1.8.1.0	2.356
STUN	STUNTMAN 1.2.5	219	STUNTMAN 1.2.5	172

[1] Client Side
[2] Server Side
[3] Updated Version
[4] Emulator
[5] Track
[6] Sandbox
[7] Resource Section

www.SID.ir

Archive of SID

۳-۵- نتایج آزمایش‌ها

برای هر پیام ارزیابی‌شده، تعداد فیلدهایی را که معانی آنها به‌درستی توسط روش پیشنهادی ما و Dispatcher شناسایی شده است، شمردیم؛ نتایج در جدول (۲) نشان داده شده است. روش پیشنهادی ما توانسته است $47/62 = 75\%$ از معانی را شناسایی کند؛ درحالی‌که این نتیجه با استفاده از Dispatcher، $14/62 = 22\%$ است و برتری روش ما را در استخراج معانی فیلدها نشان می‌دهد. در ادامه جزئیات بیش‌تری از آزمایش‌های خود را ارائه می‌دهیم.

(جدول-۲): نتایج شناسایی معنای فیلدها.

(Table-2): The results of field semantics identification.

پروتکل	نوع پیام	تعداد فیلدها	تعداد فیلدهایی که معانی آنها به درستی تشخیص داده شده است - روش ما	تعداد فیلدهایی که معانی آنها به درستی تشخیص داده شده است - Dispatcher
DNS	Query	۱۵	۱۰	۷
eDonkey	Hello	۲۴	۱۸	۴
	Server Status	۵	۵	۱
	Kademlia2_req	۵	۴	۰
Modbus	Query	۷	۶	۲
Stun	Binding Request	۶	۴	۰
تعدادکل		**۶۲**	**۴۷**	**۱۴**

پروتکل DNS: پیام تحت مطالعه این پروتکل، یک پرس‌وجوی DNS است که نشانی IP میزبان www.google.com را تقاضا کرده است؛ نتایج در شکل (۴) نمایش داده شده است. با استفاده از شیوه استخراج معنای ارائه‌شده در روش پیشنهادی، معانی فلگ، نوع و طبقه به‌درستی شناسایی شده‌اند، اما این معانی، با استفاده از روش پیشنهادی Dispatcher قابل شناسایی نیست؛ چون در پیش‌نمونه هیچ یک از تابع‌های شناخته‌شده، آرگومانی با عنوان فلگ یا طبقه نداریم.

پروتکل eDonkey: شکل (۵) نتایج آزمایش برنامه سمت مشتری با پیام وضعیت خدمت‌کار[1] را نشان می‌دهد. همان‌گونه که دیده می‌شود، فرمت پیام به‌طور کامل و صحیح استخراج شده است. برای استخراج فیلدهای تعداد کاربران و فایل‌ها، خروجی ابزار Resource tuner و اطلاعات پیش‌نمونه تابع sendmessage را ترکیب کردیم.

[1] Server Status Message

نتایج ارزیابی پیام kademlia2_req با مشاهده برنامه سمت مشتری، در شکل (۶) نمایش داده شده است. شناسه گیرنده با مشاهده اطلاعات نمایش داده‌شده در واسط گرافیکی شناسایی شد. درحالی‌که با استفاده از روش ما محدوده همه فیلدها و معانی بیش‌تر آنها به‌درستی تشخیص داده شده، در شرایط مشابه Dispatcher فقط توانسته است، یکی از واژه‌های کلیدی پروتکل را شناسایی کند؛ به‌علاوه، دو فیلد شانزده بایتی را به‌اشتباه به‌عنوان هشت فیلد چهاربایتی اعلام کرده است. روش ما با استفاده از اطلاعات محیطی توانسته است، شناسه گیرنده پیام را شناسایی و محدوده فیلدها را نیز به‌درستی شناسایی کند.

پروتکل Modbus: Modbus پروتکلی است که در شبکه‌های اسکادا مورد استفاده قرار می‌گیرد. پیام پرس‌وجوی این پروتکل را با مشاهده برنامه شبیه‌ساز Ananas ارزیابی کرده‌ایم. در شکل (۷) نتایج این آزمایش نشان داده شده است. فیلد طول با استفاده از روش پیشنهادی ما برای شناسایی فیلد طول در سرآیند قابل شناسایی است؛ طول سرآیند در برنامه عدد شش مشخص شده است. Dispatcher نیز این فیلد را به‌عنوان فیلد طول شناسایی می‌کند، زیرا به‌عنوان آرگومان تابع recv استفاده شده است.

پروتکل STUN: نتایج ارزیابی پیام Binding Request در شکل (۸) نشان داده شده است. Wireshark، فیلدی با طول دوازده و معنای شناسه پیام شناسایی کرده است، البته در برخی پیاده‌سازی‌ها، شناسه پیام شانزده بایتی است و چهار بایت پیش از آن نیز متعلق به این فیلد است؛ اختلاف حاصل از نتایج ما و Wireshark به این دلیل است؛ پیداکردن چنین اختلافاتی میان پیاده‌سازی‌ها برای برخی کاربردهای امنیتی مانند تولید اثر انگشت، با اهمیت است [36]. طبق روش پیشنهادی ما بایت‌های دوم و سوم، فیلد طول را نشان می‌دهند که طول قسمت داده بعد از سرآیند را مشخص می‌کند. طول قسمت سرآیند در برنامه، بیست بایت مشخص شده است.

۶- مقایسه

در این بخش به مقایسه روش پیشنهادی و روش‌های پیشین پرداخته می‌شود. برتری اصلی روش پیشنهادشده استخراج دقیق‌تر و بیش‌تر اطلاعات فیلدهای پیام پروتکل است.

در مورد استخراج محدوده فیلدها روش پیشنهادشده، روش‌های پیشین را تکمیل کرده است. به‌عنوان نمونه در [1]

www.SID.ir

Archive of SID

Offset	Wireshark Result	Dispatcher Result	Our Result
0	Transaction ID		
2	Flags		"Msg→ flags" - Type:0x0100: query Other types: iquery, notify, update, unknown opcode
4	Questions: 1		
6	Answer PRS: 0		
8	Authority PRS: 0		
10	Additional PRS: 0		
12	Label length: 3	Length	Length:3
13	Label name: www	Name: www	Name: www
16	Label length: 6	Length	Length:6
17	Label name: google	Name: google	Name: google
23	Label length:3	Length	Length:3
24	Label name: com	Name: com	Name: com
27	Separator	Separator	Separator
28	Type: A (Host address)		Type: 0x1 Other types: 0xff,0x8001,0x2b,0x2e,0x30, 0x18
30	Class: IN(0x0001)		"Class". Warning message: message class could not be determined
32			

(شکل-۴): نتایج آزمایش پروتکل DNS – پیام Query.

(Figure-4): Experimental Results for the DNS Query message.

Offset	Wireshark's result	Dispatcher's result	Our results
0	Protocol: eDonkey(0xc3)		Type: 0xE3 Other Types: 0xC5, 0xD4
1	Message length	Third Parameter of RtlAllocateHeap function: size	Length
5	Message Type: Server Status (0x34)		Type:0x34 ServerMsgop-Server Status Other Types: 0x35-0x38-0x40-0x42-0x44
6	Number of Users:1		Users (subitem of a list in emule GUI)
10	Number of files: 0		Files (subitem of a list in emule GUI)
14			

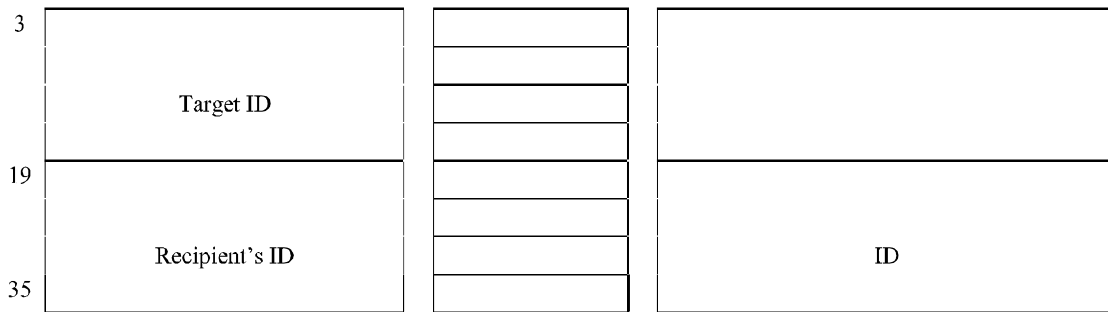
(شکل-۵): نتایج آزمایش پروتکل eDonkey– پیام Server Status.

(Figure-5): Experimental Results for the eDonkey Server Status message.

Offset	Wireshark's result	Dispatcher's result	Our results
0	Protocol: Kademlia(0xE4)	keyword	Type: 0xE4 (kad packet) Other Types: 0xC5 (emule packet), 0xD4, 0xE5, 0xB2, 0xA3
1	Message Type: KADEMLIA2_REQ (0x21)		Type: KADEMLIA2_REQ (0x21)
2	Request Type: Find node (0x0b)		Type- Message : Note: received wrong type

www.SID.ir

Archive of SID



(شکل-۶): نتایج آزمایش پروتکل eDonkey- پیام Kademlia2_req.

(Figure-6): Experimental Results for the eDonkey Kademlia2_req message.

	Wireshark Result	Dispatcher Result	Our Result
0	Transaction ID		
2	Protocol Identifier		Protocol Identifier Error Message: Bad Protocol Identifier
4	Length	Third parameter of recv function: size	Length
6	Unit identifier		UID Error Message: illegal UID
7	Function code		Function type: 0x4 Other types: 0x3, 0x10, Error Message: illegal function
8	Data bytes		Data address Error Message: illegal data address
10	Word count	Length of send data Third Parameter of send function: size	Length of send data Parameter of send function Third Parameter of send function: size
12			

(شکل-۷): نتایج آزمایش پروتکل Modbus- پیام Query.

(Figure-7): Experimental Results for the Modbus Query message.

	Wireshark Result	Dispatcher's result	Our Result
0	Message Type (Binding Request)		Type: 0x0001 Other Possible Values For This Field 0x01,0x10,0x100,0x110
2	Message Length		Length
4	Message cookie	keyword	
8	Message Transaction ID		
20	Attribute type		Type:0x3 (0x8028, 0x8)
22	Attribute Length		Length
24	Flags:00		
28			

(شکل-۸): نتایج حاصل از آزمایش پروتکل STUN – پیام binding request.

(Figure-8): Experimental Results for the STUN binding request message.

محدوده برخی فیلدها به‌اشتباه به محدوده‌های کوچک‌تری شکسته است؛ درحالی‌که روش پیشنهادی ما این محدوده‌ها را به‌درستی تشخیص داده است. فیلد طول یکی از فیلدهای اصلی و متداول در پیام پروتکل‌ها است. روش‌های ارائه‌شده در مقاله‌های پیشین [1,12,14] در برخی موارد معنای فیلد طول را به‌درستی تشخیص نمی‌دهند؛ اما با استفاده از روش

www.SID.ir

Archive of SID

پیشنهادشده در این مقاله این موارد نیز قابل شناسایی هستند.

برای بهبود نتیجه و استخراج معانی راه‌کار [10] فیلدها را در اطلاعات محیطی مانند نام کاربری، زمان‌ها، نشانی میزبان و غیره جستجو می‌کند؛ اما درصورتی‌که فیلدی در اطلاعات محیطی دیده نشود، معنای فیلد استخراج نخواهد شد. برای نمونه فیلد class در پروتکل DNS را در نظر بگیرید، با توجه به اطلاعات محیطی معنای این فیلد به‌دست نخواهد آمد؛ درصورتی‌که با استفاده از روش پیشنهادی ما این معنا به‌درستی شناسایی شده است.

در مقایسه با روش‌های مبتنی بر شبکه، روش‌های مبتنی بر برنامه با توجه به اطلاعات بیش‌تری که برنامه پیاده‌ساز پروتکل در اختیار دارد، می‌توانند اطلاعات بیش‌تری در مورد معانی فیلدها در اختیار بگذارند.

راه‌کار ارائه‌شده در مقاله [1,14] از اطلاعات آرگومان‌های توابع برای شناسایی معانی فیلدها استفاده کرده است. با استفاده از این روش می‌توان معانی گسترده‌تری نسبت به روش‌های مبتنی بر ترافیک به‌دست آورد. با این وجود، درصورتی‌که فیلدی به‌عنوان آرگومان توابع شناخته‌شده استفاده نشود، معنای آن قابل استخراج‌شدن نیست. به‌علاوه در برخی موارد حتی درصورتی‌که فیلد به‌عنوان آرگومان توابع استفاده شود، معنایی فراتر دارد. برای نمونه فیلد protocol identifier در پروتکل Modbus به‌عنوان آرگومان هیچ تابعی استفاده نشده است و حتی درصورتی‌که به‌عنوان آرگومان استفاده می‌شد، آرگومان هیچ یک از توابع استاندارد موجود چنین معنایی ارائه نمی‌دهد.

روش پیشنهادی ما به این نکته توجه کرده که در پیاده‌سازی پروتکل، اطلاعات معنایی بسیاری از فیلدها قرار گرفته است؛ به‌دلیل اینکه برنامه‌های لایه کاربرد اغلب طوری پیاده‌سازی می‌شوند که قابلیت تعامل با انسان را داشته باشند و با درنظرگرفتن این نکته می‌توان معانی گسترده‌ای از فیلدهای پیام را که مدنظر پیاده‌ساز پروتکل بوده است، استخراج کرد. درصورتی‌که در یک پیاده‌سازی از پروتکل معنای یک فیلد استخراج نشد، می‌توان از سایر پیاده‌سازی‌های آن پروتکل برای رسیدن به معنا استفاده کرد.

برای استخراج انواع مختلف فیلد نوع در مقالات پیشین، روشی ارائه نشده است. با توجه به اهمیت شناسایی این فیلد و مقادیر مختلف آن، در این مقاله برای تشخیص فیلد نوع و مقادیر ممکن آن از مزایای تحلیل پویا در کنار تحلیل ایستای برنامه استفاده شده است.

۷- نتیجه‌گیری

در این مقاله با استفاده از ترکیب تحلیل ایستا و پویای دودویی برنامه پیاده‌ساز پروتکل، سامانه جدیدی برای استخراج کامل‌تر و دقیق‌تر ساختار نحوی پیام و معانی فیلدها ارائه شد. در سامانه پیشنهادی ابتدا فیلدهای طول و جداساز پیام و سپس محدوده فیلدها شناسایی شدند. همچنین روشی برای شناسایی فیلد نوع که یکی از پراستفاده‌ترین فیلدها در پیام‌ها است، ارائه شد. در شیوهٔ ارائه‌شده برای استخراج معانی، از اطلاعات معنایی ارزشمندی که توسط برنامه برای ارتباط با کاربر فراهم شده است، استفاده شد.

برای ارزیابی سامانه پیشنهادی، چهار پروتکل DNS، eDonkey، Modbus و STUN تحلیل شدند. در مقایسه با نتایج حاصل از یک پژوهش مشابه که از پژوهش های قابل توجه اخیر است، برتری نسبی روش پیشنهادی نشان داده شد. در آینده قصد داریم، روش ارائه‌شده را برای استخراج ماشین حالت پروتکل گسترش دهیم.

۸- مراجع / 8- References

[1] J. Caballero, D. Song, “Automatic protocol reverse-engineering: Message format extraction and field semantics inference”, *Computer Networks*, vol. 57(2), pp. 451-474, 2013.

[2] M. Beddoe, “The Protocol Informatics Project”, in Toorcon, 2004.

[3] W. Cui, J. Kannan, H. J. Wang, “Discoverer: Automatic Protocol Reverse Engineering from Network Traces”, in *Proceedings of the USENIX Security Symposium*, Boston, MA, pp. 199-212, August 2007.

[4] Y. Wang, X. Yun, and M. Z. Shafiq, L. Wang, A. X. Liu, Z. Zhang, D. Yao, Y. Zhang, L. Guo, “A semantics aware approach to automated reverse engineering unknown protocols”, in *20th IEEE International Conference on Network Protocols (ICNP)*, pp. 1-10, October 2012.

[5] F. Pan, Z. Hong, Y. Du, L. Wu, “Efficient Protocol Reverse Method Based on Network Trace Analysis”, *International Journal of Digital Content Technology and its Applications*, vol.6(20), 201, November 2012.

[6] G. Bossert, F. Guihéry, and G. Hiet, “Towards automated protocol reverse engineering using semantic information”, in *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pp. 51-62, ACM, June 2014.

[7] Luo X, Chen D, Wang Y, Xie P, “A Type-Aware Approach to Message Clustering for Protocol

www.SID.ir

Archive of SID

[18] "TEMU: The BitBlaze Dynamic Analysis Component", Available: http://bitblaze.cs.berkeley.edu/temu, [Accessed: March 2019].

[19] "Qemu: Open source processor emulator" Available: http://wiki.qemu.org, [Accessed: March 2019].

[20] "Buster Sandbox Analyser", Available: http://bsa.isoftware.nl/, [Accessed: March 2019].

[21] "IDA Pro", Available: https://www.hex-rays.com, [Accessed: March 2019].

[22] "Zynamics, Binnavi- binary code reverse engineering tool", Available: http://www.zynamics.com/binnavi.html, [Accessed: March 2019].

[23] "Resource Tuner 2.04 - Resource Editor for EXE and DLL Resource Files. Edit Icon Resources, Replace Strings, Change Bitmaps", Available: http://restuner.com/, [Accessed: March 2019].

[24] "Wireshark·Go Deep", Available: https://www.wireshark.org , [Accessed: March 2019].

[25] J. Caballero, D. Song, "Automatic protocol reverse-engineering: Message format extraction and field semantics inference", Computer Networks, 57(2), pp. 451-474, 2013.

[26] M. Beddoe, "The Protocol Informatics Project", in Toorcon, 2004.

[27] W. Cui, J. Kannan, H. J. Wang, "Discoverer: Automatic Protocol Reverse Engineering from Network Traces", in *Proceedings of the USENIX Security Symposium*, Boston, MA, pp. 199-212, August 2007.

[28] Y. Wang, X. Yun, and M. Z. Shafiq, L. Wang, A. X. Liu, Z. Zhang, D. Yao, Y. Zhang, L. Guo, "A semantics aware approach to automated reverse engineering unknown protocols", in *20th IEEE International Conference on Network Protocols (ICNP)*, pp. 1-10, October 2012.

[29] F. Pan, Z. Hong, Y. Du, L. Wu, "Efficient Protocol Reverse Method Based on Network Trace Analysis", International Journal of Digital Content Technology and its Applications, 6(20), 201, November 2012.

[30] G. Bossert, F. Guihéry, and G. Hiet, "Towards automated protocol reverse engineering using semantic information", in *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pp. 51-62, ACM, June 2014.

[31] Luo X, Chen D, Wang Y, Xie P, "A Type-Aware Approach to Message Clustering for Protocol Reverse Engineering", Sensors 19, no. 3, p.716, January 2019.

[32] S Kleber, H Kopp, F Kargl, "NEMESYS: Network Message Syntax Reverse Engineering by Analysis of the Intrinsic Structure of Reverse Engineering", Sensors 19, no. 3, p.716, January 2019.

[8] S Kleber, H Kopp, F Kargl, "NEMESYS: Network Message Syntax Reverse Engineering by Analysis of the Intrinsic Structure of Individual Messages", in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, 2018.

[9] F Meng, C Zhang, and G Wu, "Protocol reverse based on hierarchical clustering and probability alignment from network traces", in *2018 IEEE 3rd International Conference on Big Data Analysis (ICBDA)*, pp. 443-447, IEEE, March 2018.

[10] K.S. Shim, Y.H. Goo, M.S. Lee, H Hasanova, M.S. Kim, "Inference of network unknown protocol structure using CSP (Contiguous Sequence Pattern) algorithm based on tree structure", in *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*, pp. 1-4, IEEE, April 2018.

[11] M.S. Lee, K.S. Shim, Y.H. Goo, M.S. Kim, "A Study on the Method to Extract Clear Fields From the Private Protocol", in *2018 International Conference on Information and Communication Technology Convergence (ICTC)*, pp. 1397-1402, IEEE, October 2018.

[12] J. Caballero, H. Yin, and Z. Liang, D. Song. "Polyglot: Automatic extraction of protocol message format using dynamic binary analysis", in *Proceedings of the 14th ACM conference on Computer and communications security,* Alexandria, VA, pp. 317-329, October 2007.

[13] Z. Lin, X. Jiang, and D. Xu, X. Zhang, "Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution", in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Vol. 8, pp. 1-15, February 2008.

[14] F. Pan, L. F. Wu, and Z. Hong, H. B. Li, H. G. Lai, C. H. Zheng, "Icefex: Protocol Format Extraction from IL-based Concolic Execution", KSII Transactions on Internet & Information Systems, 7(3), 2013.

[15] Y. Yang, K. McLaughlin, and S. Sezer, T. Littler, E. G. Im, B. Pranggono, H. F. Wang, "Multiattribute SCADA-specific intrusion detection system for power networks", in *IEEE Transactions on Power Delivery*, 29(3), 1092-1102, 2014.

[16] V. Yegneswaran, J. T. Giffin, and P. Barford, S. Jha, "An architecture for generating semantics-aware signatures", in *Proceedings of Usenix Security Symposium 2005*, pp. 34-43, August 2005.

[17] A. Aho, M. Lam, R. Sethi, and J. D. Ullman, "Compilers: Principles, techniques, and tools", 2nd ed., Addison Wesley, Boston, 2006.

www.SID.ir

Archive of SID

[43] "Qemu: Open source processor emulator" Available: http://wiki.qemu.org, [Accessed: March 2019].

[44] "Buster Sandbox Analyser", Available: http://bsa.isoftware.nl/, [Accessed: March 2019].

[45] "IDA Pro", Available: https://www.hex-rays.com, [Accessed: March 2019].

[46] "Zynamics, Binnavi- binary code reverse engineering tool", Available: http://www.zynamics.com/binnavi.html, [Acce-ssed: March 2019].

[47] "Resource Tuner 2.04 - Resource Editor for EXE and DLL Resource Files. Edit Icon Resources, Replace Strings, Change Bitmaps", Available: http://restuner.com/, [Accessed: March 2019].

[48] "Wireshark·Go Deep", Available: https://www.wireshark.org , [Accessed: March 2019].

نیره مؤمنیان مدرک کارشناسی مهندسی فناوری اطلاعات را در سال ۱۳۹۲ از دانشگاه اصفهان و مدرک کارشناسی ارشد مهندسی فناوری اطلاعات گرایش امنیت اطلاعات را در سال ۱۳۹۵ از دانشگاه صنعتی مالک اشتر دریافت کرد. زمینه‌های علاقه‌مندی وی امنیت نرم‌افزار، تحلیل بدافزار و جرم‌یابی دیجیتال است.

نشانه رایانامه ایشان عبارت است از:

nmomenyan@yahoo.com

بهروز ترک لادانی تحصیلات خود را در مقاطع کارشناسی و کارشناسی ارشد مهندسی کامپیوتر (نرم‌افزار) به‌ترتیب در سال ۱۳۷۵ در دانشگاه اصفهان و ۱۳۷۷ در دانشگاه صنعتی امیرکبیر به پایان رساند و سپس مدرک دکترای خود را در سال ۱۳۸۳ در رشته مهندسی کامپیوتر (سامانه‌های نرم‌افزاری) از دانشگاه تربیت‌مدرس اخذ کرد. وی از سال ۱۳۸۴ به عضویت هیئت علمی دانشگاه اصفهان درآمد و هم‌اکنون دانشیار گروه مهندسی نرم‌افزار این دانشگاه است. ایشان همچنین عضو پیوسته انجمن رمز ایران و نماینده انجمن در دانشگاه اصفهان است. زمینه‌های پژوهشی مورد علاقه ایشان مشتمل بر مدل‌سازی و تحلیل امنیت، توصیف و وارسی صوری، اعتماد محاسباتی، امنیت نرم‌افزار و تحلیل بدافزار است.

نشانه رایانامه ایشان عبارت است از:

ladani@eng.ui.ac.ir

Individual Messages", *in12th USENIX Workshop on Offensive Technologies (WOOT 18)*, 2018.

[33] F Meng, C Zhang, G Wu, "Protocol reverse based on hierarchical clustering and probability alignment from network traces", in *2018 IEEE 3rd International Conference on Big Data Analysis (ICBDA)*, pp. 443-447, IEEE, March 2018.

[34] K.S. Shim, Y.H. Goo, M.S. Lee, H Hasanova, M.S. Kim, "Inference of network unknown protocol structure using CSP (Contiguous Sequence Pattern) algorithm based on tree structure", in *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*, pp. 1-4, IEEE, April 2018.

[35] M.S. Lee, K.S. Shim, Y.H. Goo, M.S. Kim, "A Study on the Method to Extract Clear Fields From the Private Protocol", in *2018 International Conference on Information and Communication Technology Convergence (ICTC)*, pp. 1397-1402, IEEE, October 2018.

[36] J. Caballero, H. Yin, and Z. Liang, D. Song. "Polyglot: Automatic extraction of protocol message format using dynamic binary analysis", in *Proceedings of the 14th ACM conference on Computer and communications security,* Alexandria, VA, pp. 317-329, October 2007.

[37] Z. Lin, X. Jiang, and D. Xu, X. Zhang, "Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution", in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Vol. 8, pp. 1-15, February 2008.

[38] F. Pan, L. F. Wu, and Z. Hong, H. B. Li, H. G. Lai, C. H. Zheng, "Icefex: Protocol Format Extraction from IL-based Concolic Execution", KSII Transactions on Internet & Information Systems, 7(3), 2013.

[39] Y. Yang, K. McLaughlin, and S. Sezer, T. Littler, E. G. Im, B. Pranggono, H. F. Wang, "Multiattribute SCADA-specific intrusion detection system for power networks", in *IEEE Transactions on Power Delivery*, 29(3), 1092-1102, 2014.

[40] V. Yegneswaran, J. T. Giffin, and P. Barford, S. Jha, "An architecture for generating semantics-aware signatures", in *Proceedings of Usenix Security Symposium 2005*, pp. 34-43, August 2005.

[41] A. Aho, M. Lam, and R. Sethi, J. D. Ullman, "Compilers: Principles, techniques, and tools", 2nd ed., Addison Wesley, Boston, 2006.

[42] "TEMU: The BitBlaze Dynamic Analysis Component", Available: http://bitblaze.cs.berkeley.edu/temu, [Accessed: March 2019].

www.SID.ir