The Impact of Security Mechanisms on Software Vulnerabilities

M. Naghavi[1], T. Nooshifard[2], M. Gholami[3*]
1,2,3 - Imam Hossein University
(Receive: 2013/09/27, Accept: 2014/03/12)

Abstract

Nowadays, the vulnerabilities of operating systems and commonly applied software have provided a major hole for intruders to attacks to the information technology infrastructures. In this way, attackers may take the control of computer systems. Researches in software area have struggled a lot to develop and operate security mechanisms, to be applied in software development lifecycle, to resist the increasing growth of vulnerabilities. This paper evaluates the measure of impact and effectiveness of these security mechanisms by reviewing and statistical analysis of the existent reports in global database vulnerabilities. According to our conducted surveys, it is clarified that despite of the growing development of security mechanisms, some of the vulnerabilities have grown up and some of them are about removal from the list of the proposed vulnerabilities. The valuable point of this survey is introducing the rate of using a software as a parameter for estimating the amount of damage caused by software vulnerabilities.

Keywords:

Software, Security Mechanism, Vulnerability, Number Errors, Format String Errors, Buffer Overflow Error

***Corresponding Author Email:** g8911534827@ihu.ac.ir

مجله علمی- پژوهشی "پدافند الکترونیکی و سایبری"

سال دوم، شماره ۱، بهار ۱۳۹۳، ص ۶۰-۴۹

تأثیر مکانیزم‌های امنیتی بر آسیب‌پذیری‌های نرم‌افزاری

مهدی نقوی[۱]، تقی نوشی‌فرد[۲]، مهدی غلامی[۳*]

۱، ۲، ۳ - کارشناسی ارشد کامپیوتر، دانشگاه جامع امام حسین(ع)

(دریافت: ۹۲/۷/۵ ، پذیرش: ۹۲/۱۲/۲۱)

چکیده

امروزه آسیب‌پذیری‌های موجود در سیستم‌عامل‌ها و برنامه‌های پراستفاده، شالوده حملات نفوذگران به زیرساخت‌های فناوری اطلاعات را تشکیل می‌دهد و مهاجمان از این طریق، کنترل سیستم‌های رایانه‌ای را به‌دست می‌گیرند. پژوهشگران عرصه نرم‌افزار، تلاش زیادی در ساخت و راه‌اندازی مکانیزم‌های امنیتی در چرخه حیات نرم‌افزارها برای مقابله با رشد روزافزون این آسیب‌پذیری‌ها کرده‌اند. این مقاله، با بررسی و تحلیل آماری گزارش‌های موجود در پایگاه داده جهانی آسیب‌پذیری‌ها، اقدام به ارزیابی میزان عملکرد و اثربخشی این مکانیزم‌های امنیتی نموده است. در بررسی‌های انجام‌گرفته، مشخص شد که با وجود توسعه مکانیزم‌های امنیتی، برخی از این آسیب‌پذیری‌ها رشد صعودی داشته‌اند و برخی نیز، در آستانه حذف از لیست آسیب‌پذیری‌های مطرح قرار دارند. نکته حائز اهمیت در این تحقیق، دخالت دادن میزان استفاده از نرم‌افزارها است. با اِعمال این پارامتر بر میزان آسیب‌پذیری‌ها، میزان اثر خسارتی نرم‌افزارها تخمین زده شده و باهم مقایسه گردیده است.

واژه‌های کلیدی: نرم‌افزار، مکانیزم‌های امنیتی، آسیب‌پذیری، خطاهای عددی، خطای قالب‌رشته، خطای سرریزبافر

۱. مقدمه

گسترش نرم‌افزارها در جوامع، باعث شده است که نفوذگران، بیش از پیش در صدد سوءاستفاده از این ابزارها برآیند و پس از کشف آسیب‌پذیری موجود در نرم‌افزار، با توجه به نوع نرم افزار و نحوه ورود اطلاعات به آنها، از این آسیب‌پذیری به نفع خود بهره‌برداری می‌کنند. موسسه SANS[1] به‌همراه موسسه MITRE[2] در سال ۲۰۱۱، بیست‌وپنج خطای نرم‌افزاری که منجر به آسیب‌پذیری نرم‌افزارها می‌شوند و به‌سادگی قابل سوءاستفاده هستند را طی گزارشی بیان کرده‌اند. این گزارش برای چهار دسته از افراد از دیدگاه‌های مختلف مفید است. دسته اول، برنامه‌نویسان هستند که در تولید برنامه‌های خود، از وقوع چنین خطاهایی جلوگیری کنند. دسته دوم، استفاده‌کنندگان از نرم‌افزارها هستند تا در انتخاب نرم-افزارها از لحاظ امنیتی دقت بیشتری داشته باشند. دسته سوم، محققانی هستند که بر روی آسیب‌پذیری‌های مطرح‌شده در این گزارش، بررسی و توجه بیشتری انجام دهند و دسته چهارم، مدیران نرم‌افزاری هستند تا میزان موفقیت خود در بهبود امنیت تولیداتشان را اندازه‌گیری کنند. در این گزارش، حمله سرریزبافر که ناشی از عدم کنترل بافر دریافت‌کننده داده است، به‌عنوان سومین آسیب‌پذیری خطرناک در حوزه نرم‌افزار بیان شده است. نتایج حاصل از این حمله، اجرای کد مخرب، نقض سرویس و از دست دادن داده است. در این میان، آسیب‌پذیری قالب رشته[3]، در جایگاه بیست‌وسوم قرار دارد. هزینه لازم جهت رفع این آسیب‌پذیری کم بوده و سوءاستفاده از آن، نیاز به دانش زیادی ندارد. برای دهه های اخیر، این آسیب‌پذیری به‌عنوان آسیب‌پذیری مطرح در گزارشات امنیتی بیان شده است[۱]. راهکارهای متعددی در کنترل این آسیب‌پذیری در مراحل مختلف تولید و استفاده از نرم‌افزار ارائه شده است که می‌توان آنها را در دسته‌بندی‌های مختلفی تقسیم نمود. دسته اول، برنامه‌نویسی و طراحی امن به‌عنوان عامل انسانی تولیدکننده نرم‌افزار، دسته دوم، زبان‌های برنامه‌سازی، مترجم‌ها و ابزارهای تست نرم‌افزار به‌عنوان ابزارهای تولید نرم‌افزار، و دسته سوم، سخت‌افزار و سیستم‌عامل‌ها به‌عنوان بستر اجرای نرم‌افزار است. از

1. http://www.sans.org
2. http://cwe. mitre.org
3. Format String

* رایانامه نویسنده پاسخگو: g8911534827@ihu.ac.ir

طرف دیگر، می‌توان مکانیزم‌های دفاعی در برابر این حمله را با توجه به نوع بافر، به دو دسته حافظه دینامیک هیپ[1] و حافظه پشته[2] تقسیم کرد. در ساده‌ترین نوع این حمله که شکستن پشته[3] نام دارد، مهاجم، آدرس بازگشت را با آدرس مورد نظر خود بازنویسی می‌کند و اجرای برنامه را به محل کد مخربی که در پشته تزریق کرده است، هدایت می‌کند[۲]. چندین راه‌حلِ زمان اجرا برای جلوگیری از این آسیب‌پذیری ارائه شده است که اولین نمونه از آن، محافظ پشته[4] نام دارد. در این روش، یک مقدار خاص، بعد از آدرس بازگشت قرار می‌گیرد و قبل از بازگشت، این مقدار خاص کنترل می شود که تغییر نکرده‌باشد[۳]. پروژه پَکس[5] از جمله پروژه‌های تحقیقاتی است که در راستای امن‌سازی سیستم‌عامل لینوکس اجرا شده است. در این پروژه، روشی با هدف تصادفی کردن مشخصات فرآیندهای در حال اجرا در سطح کرنل[6] ارائه شده است که در حال حاضر، در سیستم‌عامل‌های تجاری در حال استفاده است[۴].

در این مقاله، ابتدا روند توسعه‌ای و تکاملی در مکانیزم های دفاعی مرور گردیده و سپس با استفاده از آمار استخراج‌شده از پایگاه جهانی آسیب‌پذیری[7]، تحلیلی آماری بر روند اثربخشی مکانیزم های دفاعی ارائه شده است. در نهایت، با بررسی میزان استفاده نرم‌افزارها و مقایسه آن با آمارهای آسیب‌پذیری، میزان خسارات وارده بررسی شده است.

۲. مروری بر تکنیک‌های سرریزبافر

مهاجم، پس از کشف و آشکارسازی آسیب‌پذیری در نرم افزار مورد نظر، اقدام به بازبینی و تحقیق در خصوص خطا و آسیب‌پذیری گزارش‌شده، می‌نماید. پس از بررسی‌های اولیه و بالا بودن احتمال استفاده از آسیب‌پذیری، برای اکسپلویت اقدام می‌نماید. در ادامه به برخی از روش‌های حمله سرریزبافر در پشته و هیپ اشاره می شود. در تشریح آسیب‌پذیری‌ها ، بستر اجرا، ۳۲ بیتی فرض شده است.

۱.۲. پشته

فرض کنید در بدنه تابع اصلی برنامه، یک تابع احضار شود. پس از فراخوانی تابع، اتفاقات زیر به‌ترتیب رخ می‌دهند:

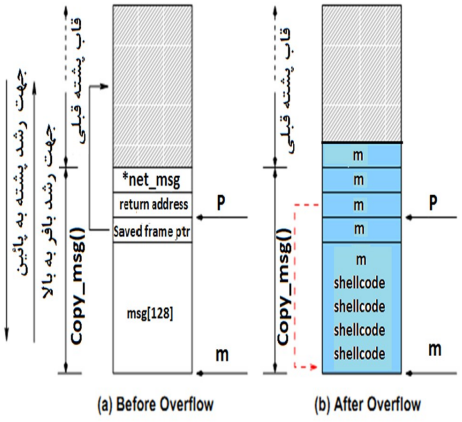
- یک قاب پشته جدید بر بالای پشته اصلی ایجاد می شود. هم‌اکنون اشاره‌گر پشته، به ابتدای پشته جدید اشاره می‌کند.
- تابع فراخوانی می‌شود. دستورالعمل فراخوانی تابع: ابتدا اشاره‌گر دستورالعمل جاری[8] را بر روی پشته قرار می دهد و سپس اشاره‌گر پشته، به اندازه ۴ بایت کاهش داده می‌شود (جهت رشد پشته، به سمت پائین فضای آدرس می‌باشد)
- اشاره‌گر، به قاب، روی پشته قرار می‌گیرد.
- مقدار اشاره‌گر پشته، در اشاره‌گر قاب پشته قرار می‌گیرد. با این عمل، قاب پشته جدید در بالای قاب پشته قبل قرار می‌گیرد.

مهاجم با استفاده از روش‌های سعی و خطا، فاصله بین اشاره‌گر پشته تا قاب پشته را به‌دست می‌آورد (این عدد برابر با طول بافر است). اگر به این عدد مقدار ۴ بایت اضافه شود، محل ذخیره سازی ثبات EIP[9] در پشته، به‌دست خواهد آمد. مهاجم با استفاده از یک رشته به طول این عدد، فضای پشته را بازنویسی می‌کند. البته ۴ بایت آخر این بازنویسی، یک آدرس در بدنه برنامه است [۵].

شکل (۱-الف)، قطعه کد آسیب‌پذیر و شکل (۱-ب)، نحوه استفاده مهاجم از قطعه کد آسیب‌پذیر در پشته را نشان می‌دهد.

```
void copy_msg(cahr* net_msg)
{
msg[128];
strcpy(msg,net_msg);
. . .
return;
}
```

شکل (۱-الف): قطعه کد آسیب‌پذیر [۶]



شکل (۱-ب): مکانیزم سرریزبافر در پشته با استفاده از کد آسیب‌پذیر [۶]

1. Heap
2. Stack
3. Stack Smashing
4. PAX Project
5. KERNEL
6. Stack Guard
7. NVD: National Vurnubility Database : nvd.nist.gov
8. Instruction Pointer (in 32bit stsytem is EIX)
۹. ثبات ذخیره‌کننده آدرس دستورالعمل جاری

مهاجم به دنبال یکی از دستورالعمل‌های زیر برای بازنویسی مقدار ذخیره‌شده ثبات EIP بر روی پشته می‌باشد:

Call Reg

Pop Ret

Push return

Jmp [Reg] +[Offset]

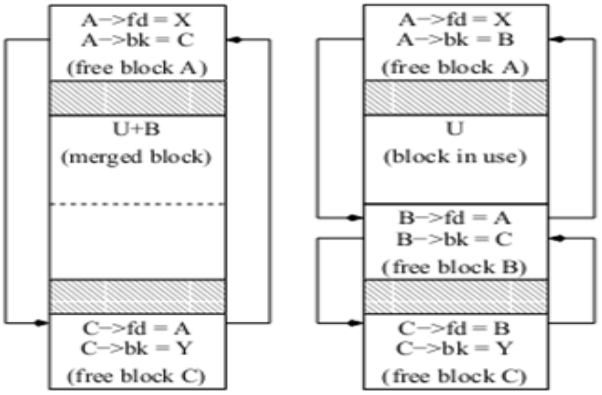
به‌صورت کلی، محتوای بافر به‌صورت زیر بازنویسی می‌شود:

آدرس یک دستور پرش + کد مخرب[۱] + سورتمه[۲] = محتوای بافر

۲.۲. حافظه هیپ

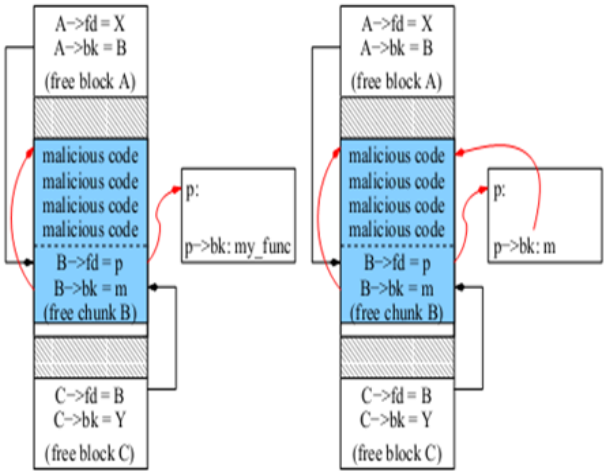
حافظه هیپ، به بلوک‌های حافظه آزاد به اندازه‌های یکسان در گروه‌های مختلف تقسیم می‌شود که هرکدام از این گروه‌ها، در یک ساختار لیست پیوندی دوطرفه مدیریت می‌شوند. این بلوک های حافظه، به‌صورت لیست دوطرفه در ارتباط با هم قرار دارند. پس از درخواست حافظه از طرف برنامه، بلوک حافظه تخصیص‌یافته‌شده، در اختیار برنامه قرار می‌گیرد. نفوذگرها از این مکانیزم جداسازی سوءاستفاده کرده و کنترل اجرای برنامه را در اختیار می گیرند. شکل ۲، نحوه قرار گرفتن سه بلوک حافظه آزاد A,B,C به‌همراه یک بلوک حافظه تخصیص‌داده‌شده Uرا نمایش می‌دهد.

وقتی بلوک حافظه U آزاد می‌شود، به‌دلیل اینکه این بلوک آزاد-شده در همسایگی بلوک آزاد B قرار دارد، باید این دو بلوک از طریق عملیاتی با یکدیگر تلفیق شوند. برای این منظور، لازم است ابتدا بلوک B از لیستی که در آن قرار دارد جدا شود. شکل ۳ روش جدا شدن بلوک B از لیست بلوک‌های آزاد را نشان می‌دهد. حال اگر حافظه U قبل از آزاد شدن توسط نفوذگر بازنویسی شود و از طریق آن، اشاره‌گرهای حافظه B نیز با مقادیر نفوذگر بازنویسی شوند، این بازنویسی با شرایطی که بیان می‌شود موجب کنترل اجرای برنامه توسط نفوذگر و انتقال اجرای برنامه به کد مخرب می شود. p، اشاره‌گر به تابعی است که نفوذگر توان فراخوانی آن را دارد. M، اشاره‌گر به کد مخربی است که توسط نفوذگر در حافظه تزریق شده است. به‌طور ساده، برای اجرای این کد مخرب لازم است که اشاره-گر p به محل کد مخرب m اشاره کند تا با فراخوانی تابعی که اشاره‌گر آن در p قرار دارد، کد مخرب اجرا شود. برای این امر، از مکانیزم جداشدن حافظه B از لیست بلوک های آزاد حافظه، سوءاستفاده می‌شود. روشی که بیان شد، یکی از روش‌های ابتدائی سوءاستفاده از سرریز حافظه هیپ می‌باشد که جزئیات آن را به طور کامل در شکل ۴ مشاهده می‌کنید [۷ و ۸].



شکل ۲. نمایش قرار گرفتن بلوک‌های حافظه

شکل ۳. نمایش جدا شدن یک بلوک حافظه B از لیست بلوک‌های آزاد



شکل ۴. نمایش نحوه قرار گرفتن و فراخوانی کد مخرب در حافظه

۳. مکانیزم‌های امنیتی در برابر حملات سرریزبافر

۱.۳. پشته

در اوایل سال ۱۹۷۲ میلادی، در زمانی که مطالعات در خصوص طراحی تکنیک‌های امنیتی کامپیوتری مورد توجه قرار گرفته بود، برای اولین بار، حملات سرریزبافر به‌صورت عمومی مستند شدند. در این گزارش، اشاره به یک رویه مانیتور شده است که کد این رویه، آدرس مبدأ و مقصد را بررسی نمی‌نماید و این باعث می‌شود تا کاربر

۳. منظور از سرسره یک زنجیره از دستورات nop می‌باشد که توسط مهاجم قبل از کد مخرب قرار می‌گیرد.

4. Sell Code

اجازه بازنویسی قسمت‌هایی از فضای رویه مانیتور را داشته باشد. در نهایت، تزریق کد به فضای مانیتور، کنترل ماشین را در اختیار مهاجم قرار می‌دهد[۹].

امروزه در این نوع حملات، فضای کرنل جایگزین فضای رویه مانیتور شده است. اولین سند تهاجمی بهره‌برداری از یک حمله سرریزبافر، در سال ۱۹۸۸ ارائه گردید. این حمله توسط کرم میرور[1] صورت گرفت و با بهره‌برداری از این حمله، توانست خود را در فضای اینترنت منتشر نماید. برنامه مورد حمله، سرویسی از سیستم عامل یونیکس به نام فینگر[2] بود [۱۰]. در سال ۱۹۹۵ میلادی، توماس لوپاتیک[3] یک سرریزبافر را کشف نموده و آن را بر روی لیست پست امنیتی باگتِرَک[4] قرار داد [۱۱]. چند سال بعد، در سال ۱۹۹۶ میلادی، الیاس لوی[5] در مجله فارک[6] مقاله‌ای تحت عنوان درهم شکستن پشته برای تفریح و کسب درآمد[7] ارئه نمود. وی معرفی گام به گام برای بهره‌برداری از آسیب‌پذیری سرریزبافر مبتنی بر پشته را منتشر نمود[۱۲] از آن زمان به بعد، دست‌کم دو نوع از کرم‌های اینترنتی از حملات سرریزبافر برای به خطر انداختن تعداد زیادی از سیستم‌ها، بهره‌برداری نمودند. در سال ۲۰۰۱ میلادی، کرم کدرِد[8] از آسیب‌پذیری سرریزبافر در سرویس اطلاعات اینترنت مایکروسافت[9] (نسخه ۵) بهره‌برداری نمود [۱۱]. در سال ۲۰۰۳ میلادی، کرم اس کیو اِل سالمِر[10]، ماشین‌هایی که سرویس اس کیو اِل سرور مایکروسافت بر روی آنها فعال شده بود را به خطر انداختند [۱۳]. سرریزبافر، در سال ۲۰۰۳، بازی‌های تحت پروانه ایکس‌باکس[11] را با حمله سرریز بافر مورد هدف قرار داد تا بتواند بازی‌های خارج از پروانه این ابزار را فعال نماید [۱۴].

کریسپین‌کوآن و همکاران، یک روش مبتنی بر کامپایلر را ارائه نمودند[۱۵]. آنان موفق شدند با قرار دادن یک مقدار تصادفی تحت عنوان کلمه‌قناری[12] قبل از آدرس بازگشت، رونویسی آدرس بازگشت ذخیره‌شده بر روی پشته را شناسائی نمایند.

بدین صورت که کامپایلر قبل از بازیابی آدرس بازگشت از روی پشته، ابتدا مقدار کلمه قناری را بررسی می‌نمود. در صورتی که مقدار کلمه‌قناری تغییر نکرده باشد، روال عادی برنامه (بازیابی آدرس بازگشت) انجام می‌گرفت. آنان این روش را به‌صورت یک وصله[13] بر روی کامپایلر لینوکسی جی‌سی‌سی[14] ارائه نمودند. کریسپین‌کوآن و همکاران همچنین به ارزیابی روش‌های محافظت از پشته پرداختند [۱۶]. کوآن روش‌های محافظت از پشته را به چهار دسته کلی تقسیم‌بندی نمود که عبارت‌اند از: نوشتن کد صحیح توسط برنامه‌نویس، بافر غیرقابل اجرا، بررسی محدوده و بررسی جامعیت اشاره‌گر. کوآن، روش جمله‌قناری را در دسته بررسی جامعیت اشاره‌گر قرار داده و به بررسی دقیق آن پرداخت و در نهایت، سربار پائین این روش را بررسی نموده است. وی در روش بررسی محدوده، ابزار پیوریفای[15] را برای بررسی مجوز اجازه دسترسی به حافظه، معرفی نموده است. واگنِر و همکاران روشی برای کشف آسیب‌پذیری‌های بالقوه در کدهای زبان C ارائه نمودند[۱۷]. آنان ابتدا به بررسی متغیرهای رشته موجود در کدهای برنامه پرداخته و سپس بافر را با استفاده از زوج محدوده بالا و پائین(محدوده متغیر) مدل نمودند. سپس با ارائه الگوریتم خاصی، اقدام به بررسی کد برنامه بر مبنای مدل ساخته شده نمودند. آنها ابزاری برای این منظور تهیه کردند که بر مبنای یک تجزیه‌گر[16] ساخته‌شده بود. لاروشل و ایوانز ابزار ال‌سی‌ال‌اینت[17] را برای شناسائی حملات احتمالی سرریزبافر از روی کد برنامه ارائه کردند[۱۸]. این ابزار با استفاده از توضیحات[18] برنامه‌نویس در خط قبل از احضار تابع، اقدام به بررسی داده‌های ارسال‌شده به تابع با استفاده از چهار پیش‌فرض اصلی برای محدوده داده‌ها[19] می‌نماید. یکی از روش‌های جلوگیری از حملات سرریزبافر، محافظت از متغیرهای ایستا و پویا است. کوآن و همکاران، یک تکنیک مبتنی بر کامپایلر برای محافظت از اشاره‌گر ارائه نمودند [۱۹]. در این روش، اشاره‌گر در هنگام ذخیره‌سازی رمز شده و در هنگام بارگذاری در ثبات پردازنده، رمزگشائی می‌شود. آنان مدعی سربار پائین این روش شدند. هاف و بی‌شاپ ابزار استوبو[20] را ارائه نمودند[۲۰]. این ابزار، کد منبع را به‌عنوان ورودی دریافت می‌نماید و سپس از روی آن، کد ثانویه‌ای تولید می‌کند که در هنگام اجرا، رفتاری مشابه با کد اولیه دارد، با این تفاوت که کد ثانویه، حاوی کدهای اضافی برای کمک به ردیابی می‌باشند. شکل (۵-الف)، یک نمونه از کدهای C را نشان می‌دهد. بازتولیدشده این کد توسط استوبو، در شکل (۵-ب) نمایش داده شده است.

1. Morris worm
2. Finger
3. Thomas Lopatic
4. BugTraq
5. Elias Levy
6. Phark
7. Smashing the Stack for Fun and Profit
8. Code Red worm
9. Internet Information Services) IIS) 5. 0
10. SQL Slammer worm
11. XBOX
12. Canary Word
13. Patch
14. Gcc Compiler
15. Purify
16. Parser
17. LCLint
18. Comment
19. minSet, maxSet, minRead , maxRead.
20. STOBO means " Systematic Testing Of Buffer Overflows "

```
void func1()
{
    char buf1[100]
    /* do stuff */
}
```

شکل (۵-الف) کد منبع قبل از بازتولید [۲۰]

یکی از وظایف شبه‌تابع مشاهده‌شده در شکل (۵. ب)، افزودن آدرس شروع به‌همراه طول بافر به لیست می‌باشد. در نهایت، لیستی از توابع مورد خطر به‌همراه سطح هشدار، به برنامه نویس ارائه می‌شود.

```
void func1() {
    char buf1[100]
__STOBO_first_stack_buf(buf1,sizeof(buf1)); }
```

شکل (۵-ب) کد منبع پس از بازتولید[۲۰]

رُویز و اِس‌لَم ابزاری به نام سیارد[1] ارائه نمودند[۲۱] که با استفاده از روش‌های بررسی محدوده دسترسی حافظه، قادر به شناسائی پویای حملات بافر بود و نحوه بررسی محدوده، با استفاده از اشیاء ارجاعی[2] صورت می‌پذیرفت. این روش برای اولین بار توسط جونز و کِلی ارائه گردید[۲۲]. در این روش، ابتدا یک ساختار حاوی آدرس مبنا و اندازه اشیاء موجود در پشته ساخته می‌شود. سپس آدرس‌های جدید قبل از قرار گرفتن در حافظه، بررسی می‌شوند تا در محدوده اختصاص‌داده‌شده در اشیاء ایجادشده از قبل، قرار نگیرند. رُویز و اِس‌لَم توانستند به‌جای استفاده از ساختار جدولی توسط جونز و کلی، از یک جدول درهم[3] استفاده نمایند تا بدین ترتیب، از سربار زمان اجرا در این روش کم کنند.

ژوزژونانن و همکاران به این منظور، روشی را ارائه نموده‌اند [۲۳]. آنها داده‌های برنامه را به سه دسته: داده‌های غیرقابل سرریز، قابل سرریز و قابل سرریز در تئوری، تقسیم نمودند. دسته اول شامل اشاره‌گرها، متغیرهای عدد صحیح، ساختارها و یونیون‌های فاقد آرایه و متغیرهای اعشاری بودند که ممکن است هدف حمله قرار بگیرند. دسته دوم داده‌هائی هستند که احتمال سرریزبافر دارند و شامل آرایه‌ای از اشاره‌گرها، ساختارها و یونیون‌های حاوی آرایه (به استثناء آرایه کاراکترها) و آرایه‌ای از ساختارها می‌باشند. دسته سوم، داده‌هایی از برنامه هستند که مورد علاقه مهاجمین برای سرریزبافر هستند، اما استفاده از آنها به‌راحتی برنامه را به سمت تزریق کد هدایت نمی‌کند و این داده‌ها توسط توابع دستکاری رشته که دارای آسیب‌پذیری هستند، مورد استفاده قرار می‌گیرند. در این روش، یک سپر محافظتی برای جلوگیری از حملات سرریزبافر در سگمنت داده‌های ایستا[4]، بین این قسمت با دیگر بلوک‌های داده‌ای برنامه، قرار داده شده است. یان و همکاران با استفاده از روش تصادفی کردن متغیرهای آرایه و اشاره‌گر در فضای برنامه، اقدام به محافظت از بافر داده مورد نظر و آدرس بازگشت در برابر حملات سرریز بافر نمودند[۲۴].

۲.۳. هیپ

تحقیقات متعددی نیز در ارتباط با روش‌های دفاعی در مقابل حمله سرریزبافر در حافظه هیپ انجام شده است که در بعضی از این روش‌ها، تمرکز بر روی تصادفی بودن آدرس اجرای کد می‌باشد. در یکی از این روش‌ها، آدرس برخی مشخصات فرآیند برنامه در حافظه، مانند مکان پشته، حافظه هیپ و کتابخانه‌های برنامه، در هنگام بارگذاری در حافظه توسط بارکننده برنامه به‌صورت تصادفی تغییر می‌کند و با این روش، امکان سوءاستفاده از آسیب‌پذیری موجود در برنامه توسط نفوذگر، تا حد قابل توجهی کاهش پیدا می‌کند[۸]. مدیریت هیپ در بسیاری از سیستم‌عامل‌ها، از مدل مدیریت حافظه هیپ داگ لی[5] پیروی می‌کنند[۲۵]. سرریز در حافظه هیپ، در مقایسه با سرریز در حافظه پشته، متفاوت است و اصلی‌ترین تفاوت آن در این است که در حافظه هیپ، الزاما ثبات EIP توسط نفوذگر بازنویسی و کنترل نمی‌شود، بلکه در بعضی موارد، با سوءاستفاده از روش تخصیص حافظه، از این آسیب‌پذیری استفاده می‌شود.

به‌منظورجلوگیری از این دسته آسیب پذیری ها، روش‌های متعددی ارائه شده است که در ادامه به برخی از آنها اشاره می‌شود. مرجع [۸]، روشی را در سطح کاربر ارائه داده است که از تصادفی کردن اجزاء مختلف برنامه، مانند: پشته برنامه، حافظه هیپ برنامه و کتابخانه‌های مرتبط به برنامه به‌منظور جلوگیری از سوءاستفاده از آسیب‌پذیری استفاده می‌کند. در این روش، بارکننده برنامه در حافظه، قبل از انتقال فرایند اجرا به برنامه، تغییرات لازم را در آن ایجاد می‌کند. این روش، تاحدی از سوءاستفاده آسیب پذیری جلوگیری می‌کند، ولی به‌دلیل آنکه این تغییرات در سطح کاربر است، فقط می‌تواند برنامه‌های سطح کاربر را در این روش امن سازد و همچنین، به‌دلیل عملیات دوباره تخصیص حافظه، سربار را بر

1. CRED
2. Referent Objects
3. Hash Table
4. Bss Segment
5. Doug Lea

اجرای برنامه تحمیل می‌کند. یکی از روش‌های شناسایی و جلوگیری از حملات، روش مبتنی بر امضاء و مشخصات حمله است. از جمله مواردی که به‌عنوان نشانه‌های حمله توسط ابزارهای شناسایی، نظیر سیستم‌های تشخیص نفوذ[1] برای حملات سرریزبافر مورد استفاده قرار می‌گیرد، لیستی از مقادیر NOP است که به اصطلاح، سورتمه[2] نامیده می‌شود[۲۶]. در این روش، به منظور حفاظت از سرویس‌دهنده‌های اینترنتی، روشی مبتنی بر این الگوی شناسایی ارائه شده است که برای نمونه، آن را در سرویس‌دهنده وب‌آپاچی[3] پیاده‌سازی کرده‌اند. این روش، جامعیت لازم به جهت اجرا شدن در شرایط و پروتکل‌های مختلف را ندارد و همچنین، نیاز به تغییرات متعدد در سیستم‌های سرویس‌دهنده دارد[۲۷].

با انجام یک عملیات، کدهای اصلی برنامه را قبل از ورود به حافظه، کد می‌کند و در هنگام اجرا، آنها را با کلیدی که قبلا کد کرده است، از کد خارج می‌کند. در صورت هرگونه تزریق کد خارجی به حافظه، برنامه برای اجرا به‌دلیل اینکه نفوذگر از کلیدی که دستورات زبان ماشین از طریق آن کد شده‌اند باخبر نیست، نمی‌تواند کد مورد نظر خود را با موفقیت اجرا کند. این تحقیق، برروی یک شبیه‌ساز پردازنده اجرا شده است و برای عملی کردن این موضوع، تولید آپکدهای[4] جدیدی برای پردازنده‌ها مورد نیاز است.

در سطح کامپایلر نیز، تغییراتی به‌منظور شناسایی و جلوگیری از وقوع سرریزبافر انجام شده است. این تحقیقات، سربار زیادی را به برنامه تحمیل می‌کنند و نیاز است که برنامه دوباره کامپایل شود [۲۸ و ۲۹]. یک روش، سطح هسته را برای افزایش میزان بهم‌ریختگی کد برنامه ارائه داده است ولی برای انجام این کار، نیاز به اطلاعات تولیدشده توسط کامپایلر در زمان کامپایل برنامه دارد. برنامه در زمان اجرای اکسپلویت مربوط به آسیب‌پذیری، به دلیل وجود مکانیزم بهم‌ریختگی کد، دچار شکست[5] می‌شود. از این ویژگی به‌منظور شناسایی حملات به سرویس‌های اینترنتی استفاده شده است[۳۰].

این روش، به‌دلیل اجرا در سطح کاربر، باعث سربار زیاد در زمان اجرا است و به‌همین دلیل، امکان استفاده بهینه از سرویس‌های موثر اینترنتی را نمی‌دهد. با توجه به روشی که قبلا در سوءاستفاده از آسیب‌پذیری بیان شد، یک راه حل، جداسازی بخش داده هیپ از بخش کنترل کننده آن است[۳۱]. به‌منظور انجام این کار، فرآیند مدیریت حافظه هیپ را به یک فرآیند مستقل از برنامه منتقل کرده و بخش کنترل‌کننده بلوک‌های هیپ را از فضای حافظه برنامه، به فضای حافظه این فرآیند منتقل کرده است. این مکانیزم، نیاز به تغییرات گسترده در مدیریت حافظه سیستم‌عامل دارد. برنامه‌هایی که دارای مدیریت حافظه مستقل هستند نیز، نیاز به بازنویسی دوباره دارند. با هدف جلوگیری از چندین آسیب‌پذیری در حوزه هیپ، یک راه‌حل ارائه شده است[۳۲].

به‌منظور استفاده از این روش، نیاز است که برنامه‌های کاربردی با کتابخانه مدیریت حافظه ارائه‌شده، دوباره کامپایل شوند. برنامه‌هایی که خروجی‌های غیرثابت تولید می‌کنند و یا به خروجی‌های آنها به شرایط محیطی، مانند زمان وابسته است، نمی‌توانند از این کتابخانه استفاده کنند. فایل‌های اجرایی در محیط لینوکس، دارای یک بخش اختیاری هستند که آدرس فراخوانی‌های سیستمی در آن قرار دارد. با اصلاح بخش مدیریت حافظه سطح هسته و این خاصیتِ فایل‌های اجرائی در محیط لینوکس، از اجرای کد تزریق‌شده به برنامه جلوگیری می‌شود[۳۳].

این روش، نیاز به وجود بخش اختیاری فایل‌های اجرائی دارد. از نمونه روش‌هایی که در اجرای مدیریت حافظه هیپ، بخش داده را از بخش کنترل مجزا کرده‌اند، می‌توان مثالی را نام برد که پیاده سازی آن، نیاز به تغییرات گسترده در مدیریت حافظه هیپ دارد [۳۴]. گاهی پژوهشگران برای جلوگیری از وقوع آسیب‌پذیری، ترکیبی از روش‌های نرم‌افزاری و سخت‌افزاری را ارائه داده‌اند[۳۵]. این روش‌ها، نیازی به تغییرات در پردازنده‌ها و همچنین افزودن Upcodeهای جدید به دستورات زبان ماشین را ندارند.

روش‌های استاتیک کشف خطا در کدهای برنامه، توانایی کشف آسیب‌پذیری‌ها در ساختارهای پیچیده، مانند چندریختی[6] و اشاره‌گرها را ندارند. برخی از تحقیقات با هدف رفع این مشکل، از الگوریتم‌های ژنتیک برای کشف آسیب‌پذیری در کدهای برنامه استفاده کرده‌اند[۳۶-۳۸]. اسپری[7] کردن حافظه باکدهای مخرب، یکی از روش‌های سوءاستفاده از آسیب‌پذیری‌های مربوط به هیپ است. این روش، بیشتر در مورد آسیب‌پذیری‌های مرورگرهای اینترنتی مورد استفاده قرار می‌گیرد. در این روش، ابتدا توسط یکی از زبان‌های برنامه‌نویسی مانند جاوااسکریپت[8]، جاوا[9]، سی‌شارپ[10] و . . .، فضای حافظه با کدهای مخرب مرورگر تخصیص داده می‌شود. سپس، با اجرای آسیب‌پذیری برنامه، اجرای آن به یکی از قطعات کد تزریق-شده به حافظه برنامه منتقل می‌شود. یکی از روش‌های دفاعی در

1. IDS
2. Sledge
3. Apache
4. Opcode
5. Crash
6. Polymorphism
7. Spray
8. JavaScript
9. Java
10. C#

برابر اسپری کردن، در مرجع است[۳۹].

این روش، بر روی مرروگر موزیلا[1] نسخه ۱۶/۰/۰/۲ پیاده‌سازی و مورد ارزیابی قرار گرفته است. در هنگام هرگونه درخواست تخصیص حافظه، محتوای فضای تخصیص‌داده‌شده توسط این ابزار مورد ارزیابی قرار می‌گیرد. در صورت داشتن مقادیری که مشابه Upcodeهای اجرایی باشد، اجرای برنامه متوقف می‌شود. یکی از ایرادات این روش، این است که اگر مقادیر مخرب، پس از عملیات بازرسی به حافظه تزریق شوند، این ابزار نمی‌تواند کدهای مخرب را تشخیص دهد. با تغییر کرنل لینوکس و همچنین کتابخانه زبان C[2]، یک راهکار مناسب در جهت جلوگیری از اسپری کردن حافظه با کدهای مخرب ارائه شده است[۴۰]. برای اجرای یک درخواست سیستمی، نیاز است که کد موجود در سطح کاربر، از طریق یکی از دستورات فراخوانی توابع سطح کرنل[3]، این عمل را انجام دهد. این مکانیزم، با محدود کردن این فراخوانی‌ها در فضای حافظه، راهکاری برای شناسایی تزریق کد مخرب به حافظه را ارائه داده است. برای عملی کردن این مکانیزم دفاعی، نیاز به تغییری در کدهای برنامه وجود ندارد، ولی پیاده‌سازی آن، نیاز به تغییرات گسترده در کرنل و کتابخانه زبان Cدارد. یکی از مواردی که توسط نفوذگران در سرریز حافظه هیپ مورد استفاده قرار می‌گیرد، بازنویسی اشاره‌گرهاست. اشاره‌گرها، از جمله موارد پرکاربرد در زبان‌های سیستمی مانند C هستند. یک راهکار برای جلوگیری از این نوع سوءاستفاده ارائه شده است[۷]. در این مکانیزم دفاعی، بخش مدیریت حافظه کتابخانه C در سیستم‌عامل لینوکس، به‌گونه‌ای تغییر داده شده است تا اشاره‌گرهای مورد استفاده در برنامه، به‌صورت رمزشده در حافظه قرار داده شوند. قبل از ارجاع و استفاده از آنها، این اشاره‌گرها از رمز خارج شده و سپس مورد استفاده قرار می‌گیرند. با این روش، درصورت بازنویسی اشاره‌گر توسط نفوذگر، به‌دلیل رمزشدن آن توسط کتابخانهC، دیگر امکان سوءاستفاده از این اشاره‌گر وجود ندارد.

۴. بررسی و تحلیل آماری آسیب‌پذیری‌ها

تاکنون مکانیزم‌های ارزیابی امنیتی در حوزه آسیب‌پذیری حافظه نرم‌افزارها، به دو صورت انجام پذیرفته است؛ روش اول، استفاده از ابزارهای مانیتورینگ و رصد کردن خروجی نرم‌افزارها در محیط آزمایشگاهی است [۴۱ و ۴۲] و روش دوم، تحلیل و ارزیابی آماری آسیب‌پذیری‌ها و مقایسه آنها با یکدیگر می‌باشد[۴۳].

در این مقاله، با استناد به گزارش‌های منتشرشده آسیب‌پذیری‌های نرم‌افزار در مرجع [۴۴]. ، به بررسی و مقایسه آنها با یکدیگر پرداخته می‌شود. این ارزیابی، در سه بخش سیستم‌عامل، مرورگر و نرم‌افزارهای کاربردی پراستفاده انجام می‌پذیرد. سپس، میزان اثربخشی مکانیزم‌های دفاعی شرح‌داده‌شده در بخش قبل، با توجه به گزارش‌های آسیب‌پذیری موجود در همان مرجع، تحلیل و بررسی می‌شود. آسیب‌پذیری‌های بررسی‌شده عبارت‌اند از: خطاهای بافر[4]، قالب‌رشته[5] و خطاهای عددی[6].

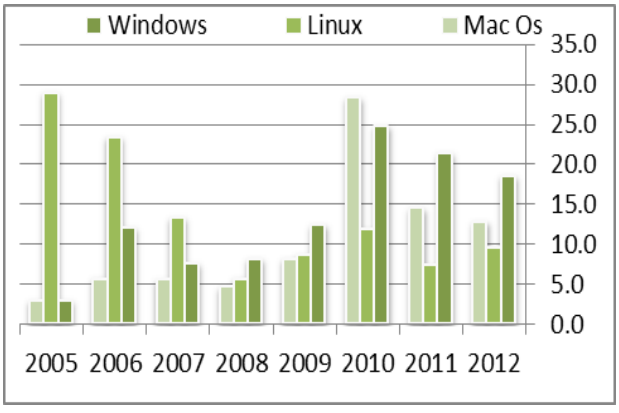
۱.۴. بررسی آسیب‌پذیری خطاهای بافر

آسیب‌پذیری خطای بافر، ناشی از عدم کنترل مناسب ورودی‌ها به برنامه است. پشته و هیپ، شاخص‌ترین محل در این نوع حملات هستند. در این بخش، آسیب‌پذیری خطاهای بافر، در سه دسته‌بندی سیستم‌عامل‌ها، مرورگرها و نرم‌افزارهای پرکاربرد، ارائه می‌شود.

۱.۱.۴. سکوی اجرا

نمودار ۱ ، بیانگر تعداد آسیب‌پذیری‌های گزارش‌شده بر اساس بستر اجرای نرم‌افزارها است. همان‌طور که در نمودار ۱ مشاهده می‌شود، بیشترین میزان آسیب‌پذیری در حوزه خطای بافر، مربوط به سیستم‌عامل‌های ویندوز و مک است. در سال‌های ۲۰۰۵ تا ۲۰۰۷، آمار بیشترین آسیب‌پذیری در این حوزه، مربوط به سیستم عامل لینوکس است. اما رشد این آسیب‌پذیری در لینوکس از سال ۲۰۰۷ تاکنون، نسبت به دیگر سکوهای اجرایی کمتر بوده است.

در مجموع، روند کلی، نشان‌دهنده رشد این آسیب پذیری در سکوهای اجرایی است. اما، سکوی اجرایی لینوکس در کنترل این آسیب‌پذیری موفق‌تر از دیگران بوده است.



نمودار ۱. درصد آسیب‌پذیری خطای بافر به تفکیک سکوی اجرا [۴۴]

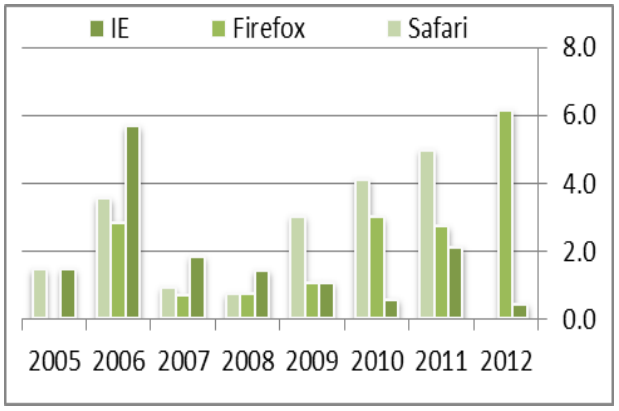
1. Mozilla FireFox
2. Glibc
3. Int80 or Sysenter

4. Buffer Error
5. Format String
6. Numeric Errors

۲.۱.۴. مرورگرها

نمودار ۲، رشد صعودی خطای بافر را در مرورگرهای معروف نشان می‌دهد. بیشترین رشد مربوط به مرورگرهای سافاری از شرکت مَکینتاش و مرورگر فایرفاکس است. رشد چشمگیر این آسیب‌پذیری در مرورگرهای سافاری و فایرفاکس، در سال‌های ۲۰۰۸ تا ۲۰۱۱ مشهود است.

در سال ۲۰۱۰، حجم آسیب پذیری‌های گزارش‌شده در مرورگرهای سافاری و فایرفاکس تا سه برابر افزایش یافته است، اما این اتفاق در سال ۲۰۱۱، برای مرورگر مایکروسافت رخ داده است. در بررسی کلی این نمودار، می‌توان به عملکرد یکسان مرورگرهای مورد بحث تا سال ۲۰۰۹ و رشد ۳ تا ۵ برابر این آسیب پذیری در سال ۲۰۱۱ برای هر سه مرورگر، اشاره نمود.



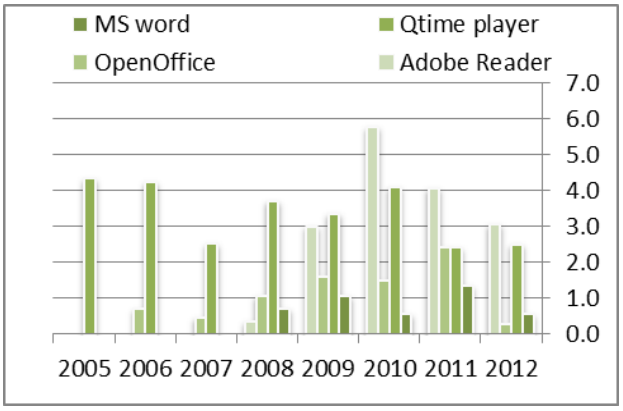
نمودار ۲. درصد آسیب‌پذیری خطای بافر در مرورگرها [۴۴]

۳.۱.۴. برنامه‌های کاربردی پراستفاده

با نگاه اولیه به نمودار ۳، مشاهده می‌شود که این آسیب‌پذیری برای ویرایش‌گر مایکروسافت به‌مراتب پایین‌تر از دیگر نرم افزارها است. این تفاوت قابل ملاحظه، در سال‌های ۲۰۰۵ تا ۲۰۱۲ مشاهده می‌شود. نرم‌افزارهای QuickTime و Adobe Reader در سال‌های ۲۰۰۹ تا ۲۰۱۲، بیشترین آسیب‌پذیری را داشته‌اند. در مجموع، رشد صعودی خطای بافر در نرم‌افزارها مشهود است، اما میزان این آسیب‌پذیری در نرم‌افزارهای چندسکوئی[1] بیشتر است.

۲.۴. بررسی آسیب‌پذیری قالب رشته

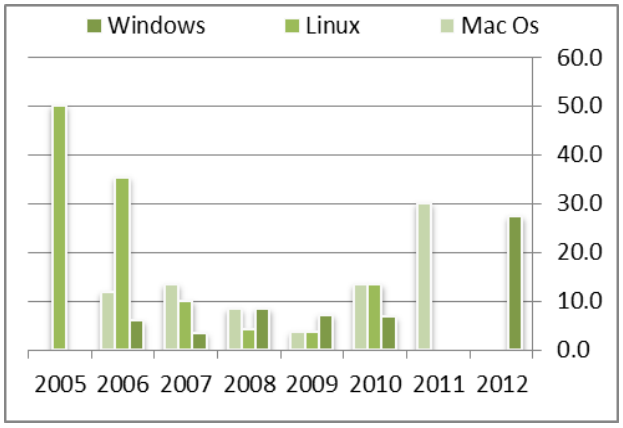
آسیب‌پذیری قالب‌رشته، ناشی از ضعف در کنترل ورودی توابع کتابخانه‌ای کار با رشته‌ها است. این بخش، شامل دسته‌بندی آماری آسیب‌پذیری قالب‌رشته سیستم‌عامل‌ها، مرورگرها و نرم افزار های پرکاربرد است.



نمودار ۳. درصد آسیب‌پذیری خطای بافر در برنامه‌های کاربردی پراستفاده [۴۴]

۱.۲.۴. سکوی اجرا

نمودار ۴، نشان‌دهنده کاهش آسیب‌پذیری قالب رشته در نرم‌افزارها با سکوی اجرائی ویندوز و لینوکس است، اما تا سال ۲۰۱۱، آسیب‌پذیری در نرم‌افزارها با سکوی اجرائی مک نسبت به سایر سیستم‌عامل‌ها، دارای رشد چشمگیری بوده است.



نمودار ۴. درصد آسیب‌پذیری قالب‌رشته به تفکیک سکوی اجرا[۴۴]

۲.۲.۴. مرورگرها و برنامه‌های کاربردی پراستفاده

این آسیب‌پذیری در حوزه مرورگرها، فقط در سال‌های ۲۰۰۷ و ۲۰۰۸ مشاهده می‌شود. در حوزه برنامه‌های کاربردی پراستفاده، در سال‌های ۲۰۰۵ تا ۲۰۱۱، هیچ موردی از این آسیب‌پذیری گزارش نشده است. این روند، حاکی از کنترل آسیب‌پذیری قالب رشته در حوزه مرورگرها و برنامه‌های کاربردی است. این موضوع، بیان‌گر سادگی کنترل این آسیب‌پذیری می‌باشد؛ زیرا فقط لازم است توابع قالب‌رشته در کتابخانه‌های مورد استفاده نرم‌افزارها، اصلاح گردند.

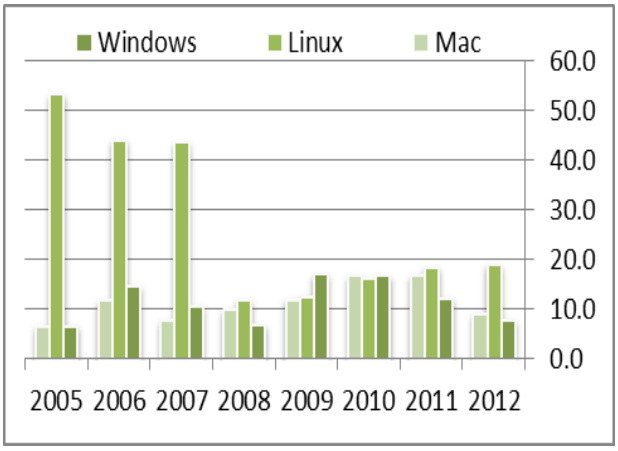
1. Multi Platform

۳.۴. بررسی آسیب‌پذیری خطاهای عددی

آسیب‌پذیری خطاهای عددی، می‌تواند منشأ حملات دیگر، از جمله خطاهای بافر باشد. از این موارد می‌توان به خطای سرریز عدد صحیح در فرآیند تخصیص حافظه اشاره نمود. این آسیب‌پذیری، ناشی از ضعف دانش مفهومی برنامه‌نویس در مورد ماهیت متغیرهای عددی است. این بخش، شامل دسته‌بندی آماری آسیب پذیری خطاهای عددی بر اساس سکوهای اجرایی، مرورگرها و نرم‌افزارهای کاربردی پراستفاده است.

۱.۳.۴. سکوی اجرا

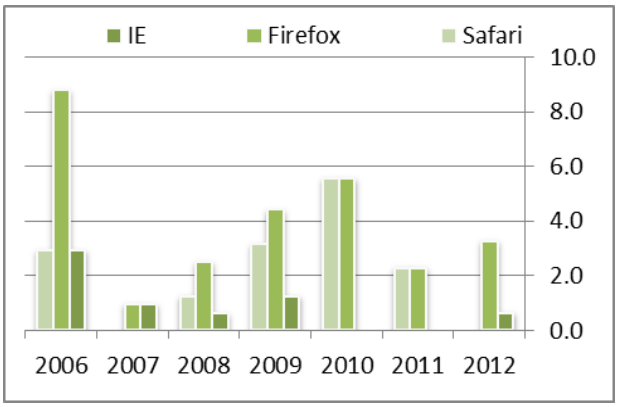
نمودار ۵، رشد صعودی آسیب‌پذیری خطاهای عددی در نرم‌افزارها بر اساس سکوی اجرا را نشان می‌دهد. سکوی اجرای لینوکس، در بین سال‌های ۲۰۰۷ تا ۲۰۱۲ نسبت به سایر سکوهای اجرا، دارای بیشترین گزارش آسیب‌پذیری خطاهای عددی بوده است. گزارش این آسیب‌پذیری در سیستم‌عامل‌های مک و ویندوز در سال‌های ۲۰۰۵ تا ۲۰۱۰، رشد صعودی داشته است.



نمودار ۵. درصد آسیب‌پذیری خطاهای عددی به تفکیک سکوی اجرا [۴۴]

۲.۳.۴. مرورگرها

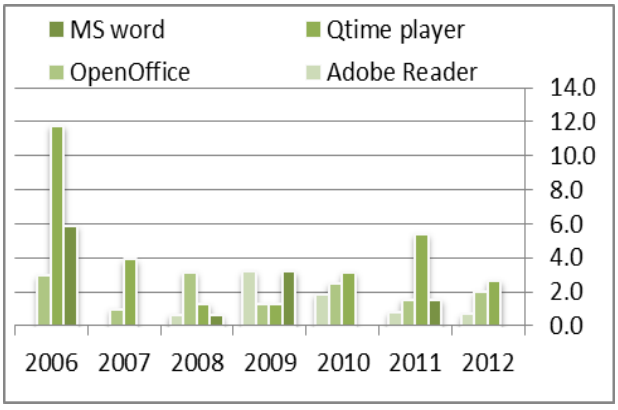
نمودار ۶، نشان‌دهنده این مطلب است که بیشترین گزارش آسیب‌پذیری خطاهای عددی در سال‌های ۲۰۰۸ تا ۲۰۱۲، مربوط به مرورگرهای چندسکوئی بوده و مرورگر مایکروسافت، در زمینه کاهش این آسیب‌پذیری موفق بوده است.



نمودار ۶. درصد آسیب‌پذیری خطاهای عددی در مرورگرها [۴۴]

۳.۳.۴. برنامه‌های کاربردی پراستفاده

نمودار ۷، نشان‌دهنده روند صعودی این آسیب‌پذیری در نرم‌افزار شرکت مکینتاش است. در این میان، نرم‌افزار چندسکوئی Adobe reader، توانسته تا حد خوبی بر این آسیب‌پذیری غلبه کند.



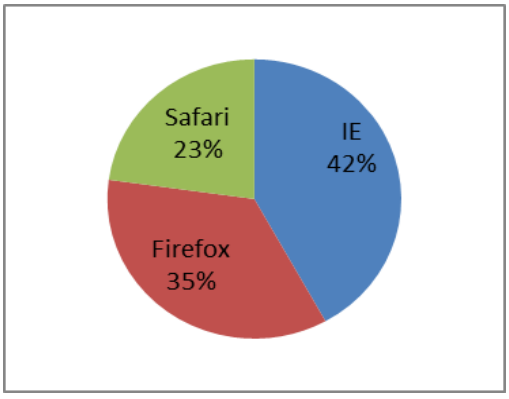
نمودار ۷. درصد آسیب‌پذیری خطاهای عددی در برنامه‌های کاربردی پراستفاده[۴۴]

ویرایش‌گر مایکروسافت و نرم‌افزار چندسکوئی Adobe Reader در سال ۲۰۰۹، بیشترین میزان آسیب‌پذیری را داشته است. آمار این آسیب‌پذیری در نرم‌افزارهای مذکور، در سال‌های ۲۰۱۰ تا ۲۰۱۲، تا حد مطلوبی کاهش یافته است، به‌طوری‌که، ویرایشگر مایکروسافت در سال ۲۰۱۰، در این خصوص گزارشی نداشته است.

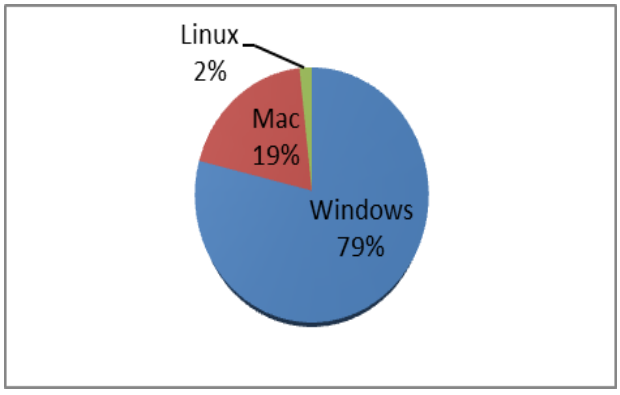
۵. بررسی تحلیلی میزان خسارت وارده به کاربران

یکی از پارامترهای موثر در تخمین میزان خسارات وارده ناشی از آسیب‌پذیری نرم‌افزارها، میزان استفاده از نرم‌افزارها است. نمودارهای ۸ و ۹ به ترتیب، درصد فراوانی استفاده از مرورگرها و

سیستم‌عامل‌های مطرح را نشان می‌دهند. با توجه به آمار بیان‌شده در قسمت قبل و آمار این نمودارها، می‌توان میزان خسارات ناشی از هرکدام از آسیب‌پذیری‌ها را به‌دست آورد. بر این اساس، طبق نمودار ۸ مشاهده می‌کنیم که در صورت وقوع یک آسیب‌پذیری در مرورگر مایکروسافت، میزان خسارات وارده تقریبا دو برابر میزان خسارات ناشی از استفاده از مرورگر سافاری است. بر طبق این مطلب و نمودار ۲، می‌توان گفت با وجود اینکه آسیب‌پذیری خطای بافر در مرورگر مایکروسافت بهتر از سافاری کنترل شده است، ولی در سال‌های اخیر، میزان خسارات در مرورگر مایکروسافت به‌مراتب بیشتر از سافاری است. این موضوع در مورد مرورگر موزیلا و سافاری نیز صادق است. با توجه به نمودار ۱ و ۹، می‌توان گفت که خطرات ناشی از آسیب‌پذیری خطای بافر در سکوی‌اجرای ویندوز، طی سال‌های اخیر به مراتب نسبت به سایر سکوهای اجرائی، منشاء خسارات بیشتری بوده است.



نمودار ۸. درصد استفاده از مرورگرها [۴۵]



نمودار ۹. درصد استفاده از سیستم‌عامل‌ها[۴۶]

با توجه به اینکه استفاده از آسیب‌پذیری قالب‌رشته پیچیدگی زیادی ندارد، لذا درصورت وقوع، می‌تواند به‌سادگی توسط نفوذگران حرفه‌ای مورد سوءاستفاده قرار گیرد. با توجه به نمودار ۴ و ۹، می‌توان گفت که در رابطه با این آسیب‌پذیری طی سال‌های اخیر، سکوی اجرایی مکینتاش منشأ خسارات بیشتری بوده است. همچنین، بر طبق نمودار ۶ و ۸، مرورگر موزیلا در ارتباط با این آسیب‌پذیری نسبت به سایر مرورگرها، در سال‌های اخیر منشأ خسارات بیشتری بوده است. با توجه به نمودارهای ۵ و ۹، می توان گفت که با وجود کنترل بیشتر خطاهای عددی، سکوی اجرایی ویندوز در سال‌های اخیر نسبت به سایر سکوهای اجرا، میزان خسارات بیشتری نسبت به سایرین داشته است.

جدول ۱. تعداد آسیب‌پذیری‌ها در سال‌های ۲۰۰۵ تا ۲۰۱۲ در یک نگاه

محدوده کاربرد	آسیب‌پذیری / عنوان	Buffer Error	Format String	Numeric Error
سکوهای اجرائی	لینوکس	۳۷۳	۱۴	۱۸۶
	مک	۴۴۶	۱۴	۱۱۱
	ویندوز	۵۷۶	۱۰	۱۱۰
مرورگرها	IE	۵۱	۱	۵
	FireFox	۹۵	۱	۳۲
	Safari	۸۶	۱	۲۰
نرم‌افزارهای پرکاربرد	Quick-Time	۱۱۶	۰	۳۳
	Ms Word	۲۶	۰	۱۰
	Open Office	۴۴	۰	۱۹
	Adobe Reader	۱۰۰	۱۱	۱۱

نمودار ۶ و ۹، بیانگر این مطلب است که مرورگر مایکروسافت در طی سال‌های اخیر، در رابطه با آسیب‌پذیری خطای عددی، دارای خسارات بسیار کمتری بوده و همچنین، بیشترین خسارات مربوط به مرورگر موزیلا بوده است. به‌طور کلی، استفاده هرچه بیشتر در یک حوزه، خسارات ناشی از آسیب‌پذیری در آن حوزه را افزایش خواهد داد. جدول ۱، تعداد آسیب‌پذیری‌ها در سه محدوده سکوهای اجرائی، نرم‌افزارهای پرکاربرد و مرورگرها در سال های ۲۰۰۵ تا ۲۰۱۲ را نمایش می‌دهد. آمار ارائه‌شده در این جدول تاییدکننده مطالب بالا است.

۶. نتیجه

در این مقاله، مدلی برای میزان اثربخشی مکانیزم‌های دفاعی بر روند کنترل آسیب‌پذیری‌ها ارائه گردید. بدین منظور، آسیب‌پذیری‌های خطای بافر، قالب رشته و خطاهای عددی در سه حوزه سکوهای اجرائی، مرورگرها و نرم‌افزارهای کاربردی پراستفاده، مورد بررسی قرار گرفتند. آمار ارائه‌شده، نشان‌دهنده این مطلب

است که جز در بعضی موارد، هرچه میزان استفاده از یک نرم افزار افزایش داشته باشد، آسیب‌پذیری گزارش‌شده نیز کاهش پیدا می‌کند. به‌عنوان نمونه، می‌توان به کاهش آسیب‌پذیری خطای عددی در مرورگر پراستفاده مایکروسافت اشاره نمود. بنابراین، طبق آمارهای ارائه‌شده، بین تعداد آسیب‌پذیری‌ها و میزان استفاده از نرم‌افزارها، ارتباط معکوس وجود دارد. از طرفی، خسارات نرم‌افزارهای پراستفاده، بالاتر است و تحقیقات مرتبط با این آسیب‌پذیری‌ها، نتوانسته‌اند میزان خسارات را تا حد قابل قبولی کاهش دهند.

۷. کارهای آینده

برخی از مکانیزم‌های دفاعی، به‌دلیل ملاحظاتی از جمله کارائی و هزینه بالای اجرا (تغییرات سخت‌افزاری و کامپایل مجدد نرم‌افزارها)، امکان استفاده در سیستم‌های تجاری را نداشته اند. می‌توان راه‌کارهای اجرای برخی ایده‌های کارآمد را در بسترهای اجرایی مختلف، مورد بررسی قرار داد.

باید به این نکته توجه داشت که نتایج کسب‌شده، فقط در مورد تعداد آسیب‌پذیری‌ها بیان شده است و می‌توان میزان پیچیدگی استفاده از آسیب‌پذیری و همچنین ضریب حساسیت آسیب‌پذیری درباره هرکدام از موارد را مورد بررسی قرار داد. در رابطه با محاسبه خسارات ناشی از استفاده مرورگرها، می‌توان میزان آسیب‌پذیری نرم‌افزارهای طرف سوم[1]، مانند فلش پلیر که در بستر مرورگرها اجرا می‌شوند را نیز لحاظ کرد.

۸. مراجع

[1] MITRE. (2011). 2011 CWE/SANS Top 25 Most Dangerous Software Errors. Available: http://cwe.mitre.org/top25/

[2] A. One, "Smashing the Stack for Fun and Profit," BugTraq Archives, p. http://immunix.org/StackGuard/profit.html, 1996.

[3] C. P. C. Cowan, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle and Q. Zhang, " Automatic Detection and Prevention of Buffer-Overflow Attacks," 7th USENIX Security Symposium, 1998.

[4] A. S. L. R. A. T.P. Team. (2003). ASLR. Available: http://pax.grsecurity.net/docs/aslr.txt

[5] WIKI. (2012). Stack buffer overflow. Available: http://en.wikipedia.org/wiki/Stack_buffer_overflow

[6] Z. K. Jun Xu, and K. I. Ravishankar , "Transparent Runtime Randomization for Security.

[7] C. P. Kyungtae-Kim, "Securing heap memory by data-pointer encoding," Elsevier, 2011.

[8] Z. K. Jun Xu, and Ravishankar K. Iyer, "Transparent Runtime Randomization for Security," 2002.

[9] J. P.anderson, "Computer Security Technology Planning Study," EDS-TR-73-51, vol. 2, pp. 61-61, Octobr 1972 1972.

[10] D. Seeley. A Tour of the Worm. Available: http://web.archive.org/web/20070520233435/http://world.std.com/~franl/worm.html#p4.5.2

[11] WIKI. (2012). Buffer overflow. Available: http://en.wikipedia.org/wiki/Buffer_overflow

[12] D. Alhambra "Smashing The Stack For Fun And Profit," phrack vol. seven, August 1996.

[13] M. technet, "Microsoft Security Bulletin MS02-039," 2007-06-03 2003.

[14] Games industry, "Hacker breaks Xbox protection without mod-chip," 2003.

[15] C. P. C. Cowan, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang,, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," 1998.

[16] P. W. C. Cowan, C. Pu, S. Beattie, and Jo. Walpole, "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade*," 1999.

[17] J. S. F. D. Wagner, E. A. Brewer, Al. Aiken, "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities," 2000.

[18] D. E. D. Larochelle, "Statically Detecting Likely Buffer Overflow Vulnerabilities," 2001.

[19] S. B. C. Cowan, J. Johansen, P. Wagle, "Point GuardTM: Protecting Pointers From Buffer Overflow Vulnerabilities," ed Washington, D.C., USA, 2003.

[20] M. B. E. Haugh "Testing C programs for buffer overflow vulnerabilities," 2003.

[21] M. L. O. Ruwase, "A Practical Dynamic Buffer Overflow Detector," 2004.

[22] P. K. R. Jones "Backwards-compatible bounds checking for arrays and pointers in C programs," pp. 13-26, 1997.

[23] F. P. Y. Younan, W. Joosen, "Protecting global and static variables from buer overflow attacks without overhead," 2006.

[24] Y. F. Y. Fen, S. Xiaobing, Y. Xinchun, M. Bing "A New Data Randomization Method to Defend Buffer Overflow Attacks," Elsevie, 2011.

[25] D. Lea. (2009). A Memory Allocator. Available: http://g.oswego.edu/dl/html/malloc.html

[26] C. K. Th. Toth, "Accurate Buffer Overflow Detection via Abstract Payload Execution," 2002.

[27] A. D. K. Gaurav S. Kc, Vassilis Prevelakis, "Countering Code-Injection Attacks With Instruction-Set Randomization," Copyright 2003 ACM, 2003.

[28] M. S. L. Olatunji Ruwase, "A Practical Dynamic Buffer Overflow Detector," 2003.

1. Third Party

[29] J. J. Ch. Kil, Ch. Bookholt, J. Xu, P. Ning, "Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software," 2005.

[30] P. N. Jun Xu, Ch. Kil, Y. Zhai, Ch. Bookholt, "Automatic Diagnosis and Response to Memory Corruption Vulnerabilities," ACM, 2005.

[31] X. J. M. Kharbutli, Y. Solihin, G. Venkataramani, M. Prvulovic, "Comprehensively and Efficiently Protecting the Heap," Intl. Symp. on Architecture Support for Programming Languages and Operating Systems, 2006.

[32] B. G. Z. Emery D. Berger, "DieHard: Probabilistic Memory Safety for Unsafe Languages," ACM, 2006.

[33] M. R. C. M. Linn, S. Baker, C. Collberg, S. K. Debray, J. H. Hartman, "Protecting Against Unexpected System Calls," 2005.

[34] W. J. Y. Younan, and F. Piessens, "Efficient protection against heap-based buffer overflows without resorting to magic," 2005.

[35] J. C. Z. Shao , K. C.C. Chan, C. Xue, E. H.-M. Sha, "Hardware/software optimization for array & pointer boundary checking against buffer overflow attacks," ScienceDirect, 2006.

[36] A. G. Del Grosso C, Di Penta M, "An evolutionary testing approach to detect buffer overflow," 2004.

[37] D. P. M. Del Grosso C, Antoniol G, Merlo E, Galinier P, "Improving network applications security: a new heuristic to generate stress testing data," 2005.

[38] G. A. C. Del Grosso, E. Merlo, P. Galinier, "Detecting buffer overflow via automatic test input data generation," www.elsevier.com, 2007.

[39] B. L. P. Ratanaworabhan, B. Zorn, "Nozzle: A Defense Against Heap-spraying Code Injection Attacks," Microsoft Research Technical Report MSR-TR-2008-176, 2008.

[40] C. H. H. Fu-Hau Hsu, Chi-Hsien Hsu, Chih-Wen Ou, Li -Han Chen, Ping-Cheng Chiu, "HSP: A solution against heap sprays," http://www.elsevier.com/locate/jss, 2010.

[41] Symantec, "Analysis of GS protections in Microsoft® Windows Vista™," 2007.

[42] Symantec, "An Analysis of Address Space Layout Randomization on Windows Vista™," 2010.

[43] Secunia, "DEP/ASLR Implementation Progress in Popular Third-party Windows Applications," 2010.

[44] nvd.nist.gov/, "National Vulnerability Database," 2012.

[45] Wiki. (2012). Usage_share_of_web_browsers. Available: http://en.wikipedia.org/wikiUsage_share_of_web_browsers

[46] Wiki.(2012).Usage_share_of_operating_systems. Available :http://en.wikipedia.org/wikiUsage share of operating systems.