سرریز شدن بافر

حملات و دفاعهایی برای اسیب پذیری دهه اخیر

چکیده سرریزهای بافمتدلترین شکل اسیب امنیتی در طی ده سال گذشته است. ضمنا، شدن حوزهٔ آسیب های نفوذ شبکه از راه دور را شامل می شود در جایی که یک کاربر اینترنت درصدد بدست آوردن کنترل نسبی میزبان است. اگر آسیب های لبریزی بافر بطور موثر حذف گردد، بخش زیادب از تهدیدهای امنیتی جدی نیز حذف می شود در این مقاله به انواع آسیب های لبریزی بافر و حملات مربوطه اشاره می کنیم و اقدامات دفاعی مختلف را بررسی می کنیم که شامل روش Stack Guard است. سپس ترکیبات روش هایی را که می توانند مشکل عملکرد سیستم های موجود را حذف کنند بررسی می کنیم در حالیکه عملکرد سیستم های موجود حفظ می گردد.

مقدمه : لبریزی بافر شایع ترین مشکل آسیب امنیتی در ده های اخیر بوده است آسیب های لبریزی بافر در حوزهٔ آسیب های نفوذ شبکه از راه دور حکمفرماست در جایی که یک کاربر اینترنت به جستجوی کنترل کامل یا نسبی یک میزبان است چون این نوع حملات هر کسی را قادر می سازد که کنترل کلی یک میزبان را به عهده بگیرد آنها یکی از جدی ترین حملات هرکسی را قادر می سازد که کنترل کلی یک میزبان را به عهده بگیرد آنها یکی از جدی ترین مولود تهدید های امنیتی محسوب می شوند حملات

لبریزی بافر یک بخش مهم از حملات امنیتی را تشکیل می دهند زیرا آسیب های لبریزی به آسانی کشف می گردند. با اینحال آسیب های لبریزی بافر در گروه حملات نفوذ از راه دور قرار می گیرند زیرا یک آسیب پذیری لبریزی بافر مهاجم را با آنچه که نیاز دارد نشان می دهد یعنی توانایی تزریق و اجرای رمز (کد) حمله. کد حملهٔ تزریق شده با برنامهٔ آسیب پذیر کار می کند و به مهاجم امکان آن را می دهد که عملکرد لازم دیگر برای کنترل رایانهٔ میزبان را در اختیار بگیرد. مثلا از بین بردن حملات بکار رفته در ارزیابی آشکار سازی تهاجم در ازمایشگاه لینکلن در ۱۹۹۸ سه مورد از نوع حملات مهندسی اجتماعی اساسی بودند که به اعتبارات کاربر حمله کردند و دو مورد از نوع لبریزی های بافر بودند و ۹ مورد از ۱۳ مورد مشاوره های CERT از ۱۹٤۸ شامل لبریزی های بافر بود و حداقل نیمی از مشاوره های CERT ۱۹۹۸ شامل اینحال لبریزی های بافر بود بررسی غیر رسمی درباره خدمت آسیب پذیری امنیتی نشان داد که تقریبا 2/3 پاسخ دهندگان عقیده داشتند که لبریزیهای بافر علت اصلی آسیب پذیری امنیتی است. آسیب پذیری های لبریزی بافر و حملات در یک سری از شکل ها می آید که ما در بخش ۲ شرح می دهیم و طبقه بندی می کنیم(2) دفاع در برابر حملات لبریزی بافر بطور مشابه در یک سری شکل ها ظاهر می شود که در بخش ۳ شرح می دهیم که شامل انواع حملات و اسیب پذیری هایی است که این دفاع ها در برابر آنها

موثر است پروژه Immunix مکانیزم دفاعی Stack Guard را تو سعه داده است که در دفاع از حملات بدون لطمه به عملکرد یا سازگاری سیستم بسیار موثر بوده است . بخش ٤ به بررسی ترکیبی از دفاع هایی می پردازد که یکدیگر را پوشش می دهند. بخش ٥ نتیجه گیری ها را شامل می گردد.

۲) آسیب پذیری های لبریزی بافر و حملات

هدف کلی بک حملهٔ لبریزی بافر، خنثی کردن عملکرد یک برنامه ممتاز است طوری که مهاجم بتواند کنترل آن برنامه را بعهده بگیرد. و اگر برنامه بقدر کافی پیشرفته باشد بتواند کنترل میزبان را انجام دهد. مهاجم به یک برنامه root ریشه حمله می کند و فورا کد مشابه با "exec(sh)" را اجرا می کند تا به یک لایه root برسد

برای انجام این هدف، مهاجم باید به دو هدف دست یابد:

کد مناسب را آرایش دهد تا در فضای نشانی برنامه باشد ۲) از برنامه برای پرش به آن کد استفاده کند و پارامترهای مناسب بداخل حافظه و رجیسترها لود شوند.

ما حملات لبریزی بافر را بر حسب حصول این اهداف دسته بندی می کنیم بخش ۲.۱ نحوه قرار گرفتن کد حمله در فضای نشانی برنامه قربانی را شرح می دهد. بخش ۲.۲ شرح می دهد که چونه مهاجم یک بافر برنامه را لبریز می کند تا حالت برنامه مجاور را تغییر دهد که جایی است که قسمت لبریزی از آنجا می آید تا باعث شد که برنامه قربانی

به کد حمله پرشی نماید بخش ۲.۳ موضوعات مربوطه را در ترکیب کردن روشهای تزریق کد از بخش ۲.۱ با روش ها ی وقفهٔ جریان کنترل از بخش ۲.۲ بحث می کند.

۲.۱ روش های آرایشی کد مناسب برای فضای نشانی برنامه : دو روش برای آرایش کد حمله برای فضای نشانی برنامه قربانی وجود دارد: تزریق یا کاربرد آن چه که قبلاً در آنجا وجود دارد.

تزریق آن: مهاجم یک رشته را بصورت ورودی برای برنامه فراهم می کند که برنامه در یک بافر ذخیره شود. رشته شامل بایت هایی است که دستورالعمل های CPU برای سکوی مورد حمله می باشند در اینجا حمله کننده از بافر های برنامه قربانی برای ذخیره کردن کد حمله استفاده می کند. بعضی موارد مربوط به این روش به این شرح است:

مهاجم مجبور نیست هر بافری را برای انجام این کار لبریز کند: بارگیری کافی کافی می تواند بداخل بافرهای کاملا سالم تزریق شود بافر می تواند در هر جایی قرار بگیرد.

روی متغیرهای خودکار روی متغیرهایی mallocell در ناحیه ایتای استاتیک (مقدار طی شده اولیه یا نشده) قبلا در آنجا وجود دارد: اغلب کد برای انجام آنچه که مهاجم می خواهد انجام دهد قبلا در فضای نشانی برنامه وجود دارد و مهاجم فقط لازم است که کد را پارامتر بندی کند و سپس باعث پرش برنامه به آن شودمثلا اگر کد حمله لازم باشد تا exel ("/bin/sh") را انجام دهد در جایی که arg یک آرگومان مکان نمایی

رشته است آنگاه مهاجم باید یک مکان نما را تغییر دهد تا به ("/bin/sh") اشاره کند و به دستور العمل های مناسب در کتابخانه libe پرش نماید.

۲.۲ راههایی که برنامه باعث پرش به کد مهاجم می شود: تمام این روشها قصد دارند تا جریان کنترل برنامه را تغییر بدهند طوری که برنامه به کد حمله پرش کند. روش اصلی عبارت اند از لبریزی یک بافر است که دارای حدود غیر موجود یا ضعیف ای است که ورودی را با هدف از بین بردن یک قسمت مجاور برنامه کنترل می کند مثلا مکان نماهای مجاور و غیره با لبریز کردن بافر، مهاجم می تواند حالت برنامه مجاور را با یک توالی تقریبا اختیاری از بایت ها بازنویسی کند که منجر به یک میان بری اختیاری از سیستم نوع C و منطق برنامه قربانی می شود در اینجا طبقه بندی کردن از نوع حالت برنامه ای است که لبریزی باقر مهاجم قصد دارد تا در ان خرابی بوجود آورد. در اصل، این حالت می تواند هر نوع حالت ای باشد. مثلا morris worn از یک لبریزی بافر در مقابل برنامه fingerel استفاده کرد تا نام یک فایل ای را از بین ببرد که fingerd اجرا می نماید. در عکل اکثر لبریزی های بافر قصد دارند تا مکان نماهای کد را خراب نمایند.

رکوردهای فعالیت: هر بار که یک تابع انحصاری می شود یک رکورد فعالیت روی دسته قرار دارد می وشد که شامل نشانی برگشت ای است که برنامه باید پرش کند وقتی که تابع وجود دارد یعنی به کد تزریق شده در بخش ۲.۱ اشاره می کند. حملاتی که رکورد

فعالیت را خراب می کنند متغیرهای خودکار جریان لبریزی نشانی ها را تحت تاثیر قرار می دهند یعنی بافرهای مطابق شکل با خراب کردن نشانی در رکورد فعالیت مهاجم باعث می شود که برنامه به کد مهاجم پرش کند وقتی که تابع قربانی نشانی برگشت را برمی گرداند. این شکل لبریزی یک سری حملات لبریزی بافر را باعث می شوند.

مکان های تابع: void (*foo) متغیر foo را اعلان می کند که از نوع مکان نما برای تابع ای است که viod را بر می گرداند. مکان نما های تابع می توانند در هر جایی تخصیص داده شوند و بنابراین مهاجم فقط باید یک بافر قابل لبریزی مجاور به یک مکان نمای تابع را در هر کدام از این نواحی پیدا کند و آن را لبریز کند تا مکان نمای تابع را پیدا کند پس از مدتیوقتی برنامه احضاری را از طریق این مکان نمای تابع انجام می دهد به محل مطلوب مهاجم حمله می کند یک مثال از این نوع حمله در مقابل برنامه snperprobe برای لینوکس ظاهر گردید

بافرهای پرش بلند: c شامل یک سیستم موسوم set ymp/longsmp است. اگر مهاجم بتواند حالت بافر را خراب کند آنگاه longJmp(buffer) به کد مهاجم پرش خواهد کرئ مانند مکان نما های تابع بافرهای lingJmp می توانند هر جایی تخصیص داده شوند بنابراین مهاجم فقط باید یک بافر قابل لبریزی را پیدا کند . یک مثال از این شکل حمله در مقابل perld.003 ظاهر گردیدحنله ابتدا یک بافر LongJmp را خراب کرد

برای بازیابی استفاده شد وقتی که لبریزی های بافر آشکار می شوند و سپس شامل مود بازیافت بود که باعث شد که مفسر perl به کد حمله پرش کند.

۲.۳

ترکیب کردن تزریق کد و روش های خراب کردن و جریان کنترل

در اینجا به بررسی موضوعات ترکیب تزریق کد مهاجم و روش های خراب رکدن جریان کنترل می پردازیم. ساده ترین شکل حمله لبریزی بافر یک روش تزریق را با یک خرابی رکورد فعال سازی در یک رشته واحد ترکیب می کند مهاجم یک متغیر خودکار قابل لبریزی را قرار می دهد و به برنامه یک رشته بلند را تغذیه می کند که بطور همزمان بافر را لبریز می کند تا رکورد فعالیت را تغییر دهد و حاوی کد حمله تزریق شده است. این سکویی برای یک حمله مطرح شده توسط levey است. چون اصلاح تخصیص یک بافر محلی کوچک بسیار متداول است یک سری حالت های آسیب پذیری کد برای این شکل حمله وجود دارند. تزریق و خرابی در یک عمل انجام نمی شود. مهاجم می تواند کد را به داخل یک بافر بدون لبریز کردن آن تزریق کند و یک بافر متفاوت را برای خراب کردن یک مکان نمای کد لبریز می کند این کار تر انجام می شود اگر بافر قابل لبریزی محدودیت هایی داشته باشد که روی آن کنترل انجام شود بنابراین بافر تا تعداد معینی از بایت ها قابل لبریزی است مهاجم حمله برای قرار دادن کد بافر آسیب پذیر

نمی باشد بنابراین کد به سادگی در داخل یک بافر متفاوت با اندازه کافی قرار داده می شود. اگر مهاجم سعی کند از کد قبلی بجای تزریق آن استفاده کند، آنها لازم است تا کد را پارامتر بندی کنند مثلا قطعه کدهایی در Libe وجود دارند (مرتبط با برنامه C) که "exel something" را انجام می دهند در جایی که "something" یک پارامتر است.مهاجم می تواند از لبریزی های بافر برای خراب کردن آرگومان استفاده کند و لبریزی بافر دیگری برای خراب کردن یک مکان نمای کد بکار می رود تا به libe در قطعه کد مناسب اشاره گردد.

۳) دفاع های لبریزی بافر. چهار روش اصلی برای دفاع در برابر آسیب پذیری های لبریزی بافر و حملات وجود دارد روشی در بخش ۳.۱ شرح داده شده است روش سیستم های عامل در بخش 3.2 شرح داده می شود که برای نواحی ذخیره برای بافرهای غیر قابل اجرا است و ا زمهاجم از تزریق کد حمله حمله جلوگیری می کند. این روش بسیاری حملات را متوقف می کند. روش کامپایلر در بخش 3.3 برای اجرا کنترل یکپارچگی روی مکان نماهای کد قبل از مرجع زدایی آنهااست. این روش حملات لبریزی بافر را غیر ممکن نمی کند ولی اکثر حملات لبریزی بافر را متوقف می کند و حملاتی که متوقف نمی شوند به سختی ایجاد می شوند و مزایای عملکرد و سازگاری زیادی دارد که در بخش 3.5 شرح داده می شود

۳) کد تصحیح نوشتن این اصطلاح در هنگام نوشتن به زبان مثلا c مطرح می شود وبویژه وقتی که نوشتن و اصلاح آن مطرح باشد با وجود سابقه طولانی درک نوشتن برنامه های آسیب پذیری بطور منظم در حال ظاهر شدن هستند بنابراین بعضی ابزارها و روش ها به توسعه دهندگان کمک کرده است که آسیب پذیری های لبریزی بافر را تشخیص دهند.

ساده ترین روش عبارت اند از greb که منبع برای احضارهای کتابخانه ای از قبیل stcpy و sprintf است که طول آرگون های آنها کنترل نمی کند نسخه های کتابخانه استاندارد c توسعه یافته است که برای این منظور بکار می رود. تیم های ممیزی که با هدف ممیزی کردن مقادیر زیادی از کد ظاهر شده اند و در جستجوی اسیب پذیری های امنتیتی رایج از قبیل لبریزی های با فرو شرایط سیستم فایل می باشند. با اینحال آسیب پذیری های لبریزی بافر قابل برسی است حتی کد از راههای مطمئن دیگری استفاده می کند می توانند حاوی آسیب پذیری ها ی لبریزی بافر باشند اگر کد حاوی یک ابتدایی باشد مثلا برنامه lprm دارای آسیب پذیری لبریزی بافر است اگر چه برای مسائل امنیتی مثا آسیب پذیری های لبریزی بافر ممیزی شده اند.برای رفع مشکل bug های باقیمانده ابزارهایی debug کردن پیشرفته ای توسعه یافته اند مانند ابزارهای تزریق عیب بطور تصادفی برای تزریق عیب بطور تصادفی برای تحقیق اجزای برنامه اسیب پذیر همچنین

ابزارهای تحلیل ای ظاهر شده اند که می توانند آسیب پذیری های لبریزی بافر بسیاری را آشکار کنند این ابزار در توسعه برنامه های امنیتی مفید هستند ولی c دستور زبان به انها اجازه نمی دهد که تضمین کلی را فراهم کنند که تمام البریزی های بافر پیدا شده کند روش های debug کردن فقط تعداد آسیب پذیری های لبریزی بافر را کمینه می کنند ولی تضمین نمی کنندکه تمام آسیب پذیری های لبریزی بافر حذف شده باشد. اقداماتی از قبیل بخش ۳.۲ تا ۳.٤ باید بکار بروند مگر اینکه شخص مطمئن باشد که تمام اسیب پذیری های لبریزی بالقوه بافر حذف شده اند.

۳.۲ بافرهای غیر قابل اجرا: مفهومی کلی برای ایجاد دیتاسگمنت از فضای نشانی برنامهٔ قربانی غیر قابل ااجرا بکار می رود و مهاجمان نمی توانند کدی را اجرا کنند که بداخل بافرهای ورودی برنامه تزریق می کنند.

بسیاری از رایانه های قدیمی تر به این صورت طراحی شدند وای اخیرا سیستم های ویندوز ms و unix برای وارد کردن کد دینامیک در داخل سگمنت ها ی دیتای برنامه طراحی شده اند که بهینه سازی های عملکرد بسیاری را پشتیبانی می کنند و شخص نمی تواند همهٔ سگمنت دیتاها را بدون قربانی کردن سازگاری برنامه غیر قابل اجرا کند ولی شخص می تواند سگمنت stack را نیز قابل اجرا کند و سازگاری اگر برنامه را فقط

کند. برای linux و Solaris این بررسی ها انجام شده اند دو مورد استثنا در مهدعط وجود دارد که کداجرایی باید روی stack قرار داده شود:

تحویل signal : Linux سیگنال های unix را procers هایی تحویل می دهد و کد را حذف م یکند تا سیگنال را روی process stack تحویا دهد و بعدا یک وقفه را باعث می شود که به کد تحویل بر روی stack پرش می کند. Stacl patch غیر قابل اجرا این امر را با مشخص کردن stack قابل اجرا در طی تحویل سیگنال ذکر می کند.

Gce Trampour نشانه هایی وجود دارند که لثز کد قابل اجرا را روی stack fvhd trampoline ها قرار می دهد. با اینحال در عمل trampoline هایی را ناتوان می کند که مشکل ای بوجود نیاورده اند آن قسمت از gcc بنظر می رسد تا مورد سوء استفاده قرار گرفته باشد.

حفاظت پیشنهاد شده توسط stack segment های غیر قابل اجرایی در برابر جملات موثر هستند که بستگی به تزریق کد حمله بداخل متغیرهای خودکار دارد ولی هیچ محافظتی در برابر سایر شکل های حمله بوجود نمی آورد. حملاتی وجود دارد که این شکل از دفاع را میان بر می زند و به یک مکان نمای کد در کدی اشاره می کند که قبلا در برنامه قرار دارد سایر حملات می توانند ایجاد شوند که کد حمله را بداخل بافرهای تخصیصی یافته تزریق می کنند

۳.۳ کنترل حدود حمله در حالی که کد تزریق بای یک حمله لبریزی بافر اختیاری است، خرابی جریان کنترل ضروری است بنابراین برخلاف بافرهای غیر قابل اجـرا محـدودیت های آرایه که توقف ها را بطور کامل کنترل می کنند را آسیب پذیری های لبریـزی هـای بافر و حملات را متوقف می کنند اگر آرایه ها نتوانند اصلا لبریز شوند انگاه لبریزی های حمله نمی توانند برای خراب کردن حالت برنامه مجاور استفاده شوند برای اجرای کنترل محدوده های آرایه تمام خواندن و نوشتن های برای ارایه ها باید کنترل شوند تا تضـمین کنند که آنها در محدودهٔ تعیین شده قرار دارند. روش مستقیم عبارت اند از کنترل کـردن تمام مرجع های ارایه است ولی اغلب امکان بکار گیری روش هـای بهینـه سـازی بـرای حذف بسیاری از این کنترل ها وجود نـدارد. روش هـای مختلفـی بـرای اجـرای کنتـرل حدود آرایه وجود دارد که در پروژه های زیر مثال زده میشوند.

۳.۳ compal compiler- برای Alpla cpu بکار می رود که یـک شـکل محـدود از کنترل حدود آرایه را انجام می دهد وقتی که check –bonds استفاده می شـود کنتـرل های حدود به شیوه های زیر محدود می شوند.

- فقط مرجع های آرایه آشکارکنترل می شوند.

– چون تمام آرایه های c به مکان نماها تبدیل می شوند وقتی که بصورت آرگومان عبور داده می شوند هیچ کنترل حدودی روی دسترسی های انجام شده توسط زیر سوال ها اجرا نمیشود.

– تابع کتابخانه ای خطرناک (( ) strcpy) معمولا با کنترل کردن حدود کامپایل نمی شوند و خطرناک باقی می مانند حتی اگر کنترل کردن حدود امکان پذیر باشد.

برای برنامه های C امکان استفاده از ریاضیات مکان نما برای دسترسی به آرایه ها وعبور آرایه ها بصورت آرگومان هایی برای توابع وجود دارد و این محدودیت ها جدّی هستند. ویژگی کنترل کردن حدود، کاربرد محدود برای رفع اشکال برنامه است و آسیب پذیری های لبریزی بافر کشف نمی باشند.

Jones & kelly3.2.2: کنترل کردن حدود آرایه ها برای c– جونز و کلی یک patch gcc را توسعه دادند که کنترل حدود آرایه ها کامل را برای برنامه های C انجام می دهد. برنامه های کامپایل شده با سایر مدل های gcc سازگار هستند، زیرا نمایش مکان نماها را تغییر نداده اند. آنها یک مکان نمای پایه را از هر عبارت مکان نما بدست می آورند و ویژگی های آن مکان نما را برای تعیین عبارت در محدوده تعیین شده بکار می برند. هزینه های عملکرد ضروری هستند. یک برنامه مبتنی بر مکان نما 30

slowdown* کاسلش ۳۰ برابر را تجربه کرد چون slowdown متناسب با کاربرد مکان نما است، که در برنامه های ویژه کاملاً متداول است

عمل کامپایلر بنظر نمی آید که کامل باشد؛ برنامه های پیچیده مانند e 1000 نمی توانند اجرا شوند هنگامی که با این کامپایلر، کامپایل می شوند، با اینحال، یک نسخه جدید از کامپایلر حفظ می شود و می تواند با بسته رمز سازی نرم افزار SSH بکار برود. آزمایشات توان عملیاتی با رمز کردن نرم افزار و کامپایلر جدید توسط SSH یک * 12 Slowdown را نشان می دهد.

پروژه purify :

3.3.3 کنترل دسترسی حافظه purify (اسم است) یک ابزار رفع اشکال (debgging) کاربرد حافظه برای برنامه c است. purify از قرار دادن کد هدف برای دسترسی حافظه استفاده می کند. پس از ارتباط با رابط purify کتابخانه ها، شخص یک برنامه قابل اجرای استاندارد را بدست می آورد که تمام مراجع آرایه آن را کنترل می کند تا مطمئن شود که آنها قانونی هستند در حالیکه برنامه های حفاظت شده توسط purify بطور معمول بدون هر نوع محیط خاصی اجرا می شوند purify واقعا بصورت یک ابزار امنیتی تولید در نظر گرفته نمی شود : حفاظت purify یک Slowdown 3 تا 5

برابر را اعمال می کند، همچنین purify به سختی ایجاد و ساخته می شود و هر نسخه تولید شده آن 5000 ارزش دارد.

3.3.4 زبانهای Type safe : تمام آسیب پذیری های لبریزی بافر از فقدان ایمنی نوع در c حاصل می شوند. اگر فقط عملیات ایمنی نوع بتوانند اجرا شوند، آنگاه امکان استفاده از ورودی خلاق بکار رفته برای foo وجود ندارد تا تغییرات دلخواه را برای متغیر bav بوجود آورد. اگر کد حساس امنیتی با نوشته شود توصیه می شود که کد به زبان ایمنی ای نوشته شود مثل Java یا ML . متأسفانه، میلیون ها خط کد سرمایه گذاری شده در سیستم های عامل موجود و برنامه های کاربردی حساس به امنیت وجود دارند و قسمت اعظم آن کد به زبان C نوشته می شود. این مقاله به روش های حفاظت کد موجود از حملات لبریزی بافر اختصاص دارد. با اینحال، (JVM) یک برنامه C است و یکی از راههای حمله به یک JVM بکارگیری حملات لبریزی بافر برای خود JVM است. بنابراین بکاربردن روش دفاعی لبریزی بافر برای سیستم هایی که ایمنی را برای زبانهای ایمنی ایجاب می نمایند ممکن است نتایج مفیدی داشته باشد.

3.4 کنترل کردن یکپارچگی مکان نمای کد- هدف از کنترل کردن مکان نمای کد- با کنترل کردن حدود تفاوت دارد. بجای تلاش برای جلوگیری از خرابی مکان نماهای کد، کنترل کردن یکپارچگی مکان نمای کد در صدد است تا مشخص کند که یک مکان نمای

کد خراب شده است قبل از اینکه مرجع زدایی شود. بنابراین در حالی که مهاجم در خراب کردن یک مکان نمای کد پیشروی می کند، مکان نمای کد خراب شده هرگز استفاده نمی شود زیرا خرابی قبل از هر کاربردی آشکار می شود. کنترل کردن یکپارچگی مکان نمای کد دارای معایبی نسبت به کنترل کردن حدودی است که بطور کامل مشکل لبریزی بافر را برطرف نمی کند لبریزی هایی که بر روی مؤلفه های حالت برنامه تأثیر می گذارند (غیر از مکان نماهای کد) هنوز موفق هستند (جدول 3 در بخش 4 را برای جزئیات مطالعه کنید) ولی مزایای زیادی بر حسب عملکرد ناسازگاری با کد موجود و تلاش اجرایی دارد که ما در بخش 3.5 به ذکر آن می پردازیم کنترل کردن یکپارچگی مکان نمای کد در سه سطح مختلف بررسی شده است. Snarskii یک اجرای ازLibc برای Free BSD توسعه داد که CPV stack را برای آشکار کردن لبریزی های بافر بررسی می نماید که در بخش 3.4.1 شرح داده می شود. این اجرا در اسمبلر کد بندی دسته بندی شد و فقط رکوردهای فعال سازی را برای توابع در داخل کتابخانه Libc حفاظت می کند، اجرای Snarskii مؤثر است و برنامه هایی را Libc ار استفاده می کنند از آسیب پذیری های درون Libc محافظت می کند ولی حفاظت را به آسیب پذیریهای در هر کد دیگری توسعه نمی دهد

Strack Guard 3.4.2 : کنترل کردن یکپارچگی رکورد فعال سازی تولید شده با کامپایلر Strack Guard یک روش کامپایلر برای فراهم کردن کنترل یکپارچگی مکان نمای کد می باشد. Strack Guard بصورت یک patch کوچک برای gcc اجرا می شود که مولد کد را برای انتشار کد برای ایجاد توابع تقویت می کند. کد تنظیم های تقویت یک کلمه (Canary) را نزدیک به نشانی برگشت بر روی stack (دسته) قرار می دهد (شکل ۲). تابع تقویت شده ابتدا کنترل میکند که ببیند که آیا کلمه Canary سالم است قبل از اینکه به نشانی اشاره شده توسط کلمه نشانی برگشت پرش نماید یا خیر. بنابراین اگر یک مهاجمی تلاش تهاجمی مانند شکل ۱ انجام دهد، حمله آشکار می شود قبل از اینکه برنامه تلاش کند تا رکورد فعال سازی خراب شده را مرجع زدایی کند. موضوع مهم برای روش Canary آن است که مهاجم از ایجاد یک Canary توسط قراردادن کلمه Canary در رشته لبریزی جلوگیری می کند. Stack Guard دو روش دیگر را برای جلوگیری از چنین امری استفاده می کند :

Terminator Conary : از نمادهای خاتمه سازی متداول برای توابع کتابخانه رشته استاندارد C تشکیل می شود یعنی O(null) ، CR ، LF و ۱- (EOF)، مهاجم نمی تواند از کتابخانه های رشته C معمولی و برای قراردادن این نمادها در یک رشته لبریزی استفاده کند زیرا توابع کپی کننده خاتمه می یابند وقتی که با این نمادها برخورد نمایند.

Random Canary : Canary یک عدد تصادفی 31 بیتی انتخاب شده از زمان آغاز برنامه است. Random Canary به آسانی حفظ می شود و به سختی حدس زده می شود و زیرا برای هیچکس افشاء نمی شود و هر بار که برنامه استارت می خورد، انتخاب میشود. کنترل یکپارچگی stack توسط stack Guard به این طریق با استفاده از quasi invariant شبه ثابت ها صورت می گیرد تا درستی تخصصی سازی های نموی (incrimultal speciulizations) تضمین گردد. یک تخصصی سازی، یک تغییر با تفکر، برای برنامه است که فقط وقتی معتبر است که شرایط معینی برقرار باشد. ما چنین وضعیتی را یک شبه ثابت (نیمه متغیر) quasi invariant می نامیم زیرا تغییر می کند ولی بندرت این کار صورت می گیرد. برای تضمین درستی، synthetix یک سری از ابزارها را برای حفاظت کردن حالت شبه ثابت ها توسعه داد.

تغییرات اعمال شده توسط مهاجمان با استفاده از روشهای لبریزی بافر می تواند بصورت تخصصی سازی های غیر معتبر در نظر گرفته شود، بویژه، حملات لبریزی بافر به شبه-ثابتی حمله می کند که نشانی برگشت یک تابع فعال نباید تغییر کند در حالی که تابع فعال است. کنترل های یکپارچگی stack Guard این شبه ثابت را تقویت می کند. نتایج تجربی نشان داده اند که stack Guard حفاظت مؤثری را در برابر حملات stack smashing فراهم میکنند در حالیکه سازگاری و عملکرد را کاملا حفظ می

نماید ما مقاومت نفوذ stack Guard را قبلا گزارش کردیم هنگامی که برنامه های آسیب پذیر مختلف را بررسی کردیم (در جدول ۱ ذکر شده است). سپس یک توزیع Linux کامل را با استفاده از stack Guard ایجاد کردیم. وقتی که حملات در برابر آسیب پذیری های در Xfree 86- 3/3,2-5 و ISOF انجام گرفت آنها را نیز آزمایش کردیم و متوجه شدیم که stack Guard بطور موفقیت آمیز این حملات را آشکار کرده و رد نموده است. این تحلیل نفوذ نشان می دهد که stack Guard در آشکار کردن و جلوگیری از حملات Stack smashing حاضر و آینده بسیار موثر است. ما نسخه ای از linux 5.1 را بر ر.وی ماشین های بسیاری در مدت یکسال داشتیم که توسط stack Guard حفاظت شده اند. این linux حفاظت شده با stack Guard بر روی رایانه شخصی Crispin Crown و بر روی Sarever فایل اشتراکی گروه ما کار می کند. این توزیع linux از سایت شبکه ما صدها بار down load شده است و 55 نفر در فهرست پستی کاربر stack Guard وجود دارند. به غیر از یک مورد استثناء، stack Guard بطور یکسان با Red laut linux 5.1 عمل کرده است. این امر نشان می دهد که حفاظت stack Guard بر روی سازگاری سیستم تاثیر نمی گذارد. ما انواع آزمایشات عملکرد برای اندازه گیری بالاسری اعمال شده توسط حفاظت stack Guard را انجام داده ایم و توسط microbench mark ها افزایش در هزینه

یک فراخوانی تابع واحد، مشاهده شده است. با اینحال macro bench mark ها بعدی بر روی سرویسهای شبکه انواع برنامه هایی که به حفاظت stack Guard احتیاج دارند) بالاسری های بسیار کمی را نشان دادند.

اولین macro bench mark ها از SSH استفاده کرد که جایگزینی های رمز شده و تایید اعتبار شده قوی ای را برای فرمانهای Berkcley r+ فراهم می کند یعنی rcp ، scp می شود. SSH از رمز سازی نرم افزار استفاده می کند و بنابراین بالاسری های عملکرد را در پهنای باند پایینی نشان می دهد. ما تاثیر پهنای باند را با استفاده از scp برای کپی کردن یک فایل بزرگ از طریق رابط Loop back شبکه اندازه گرفتیم به این شرح :

Scp big sourcc local hast : big dest

نتایج نشان دادند که stack Guard هیچ هزینه ای را برای توان عملیاتی SSH نشان نمی دهد scp ژنریک برای مدت 14.5 ثانیه (0.3 -/+) کارکرد و به یک توان عملیاتی متوسط 7S54.q KB/S (0 -/+) رسید. Scp حفاظت شده توسط stack Guard مدت 13.8 ثانیه (0.5 -/+)کارکرد و به توان عملیاتی 803.8 KB/S (48.q -/+) رسید. macro bench mark ثانویه ها، بالاسری عملکرد را در server وب Apache اندازه گرفت که یک نامزد برای حفاظت stack Guard است.

اگر A patche بتواند stack smash شود، مهاجم می توانـد کنتـرل web server را متوقف کند و به مهاجم امکان خواندن محتوای شبکه مطمئن را بدهد و تغییر یـا حـذف محتوای شبکه انجام گردد web server یک مولفه بحرانـی عملکـرد اسـت کـه مقـدار ترافیک را تعیین می کند که یک ماشین server مـی توانـد پشـتیبانی نمایـد. مـا هزینـه حفاظت stack Guard را با اندازه گیری عملکـرد Apatche بـا اسـتفاده از bench mark اندازه می گیریم. Webstone bench mark انواع جنبـه هـای یـک عملکـرد web server را اندازه می گیرد و یک با (لود load) تولید شـده از انـواع مشـتریان را شبیه سازی می کند. نتایج در جدول ۲ نشان داده می شوند. همانند با SSH عملکرد بـا و بدون حفاظت stack Guard غیر قابل تمـایز اسـت. web server حفاظـت شـده توسط web server یک مزیت بسیار کمی را برای تعداد کمـی از مشـتریان نشـان مـی دهد در حالیکه نسخه حفاظت نشـده یـک مزیـت مختصـری را بـرای تعـداد زیـادی از مشتریان نشان می دهد. در بدترین حالت Apatche حفاظت نشده دارای یک مزیـت 8 ٪ در ارتباط در هر ثانیه است اگر چـه web server حفاظـت شـده دارای یـک مزیـت مختصر می باشد. ما این واریانس ها را به نویر نسبت می دهیم و نتیجـه مـی گیـریم کـه حفاظت Safe guard هیچ تاثیر مهمی روی عملکرد web server ندارد.

Point Guard 3.4.3 : کد تولید شده با کامپایلر کنترل کردن یکپارچگی از زمانی که stack Guard ساخته شد، تنوع smashing یک برتری فاحش از حملات لبریزی بافر را ایجاد کرد. گمان می رود که این امر ناشی از بعضی از الگوهایی برای حملات stack smashing باشد که در اواخر سال ۱۹۹۵ منتشر گردید. از آن هنگام به بعد ، اکثر آسیب پذیری های stack smashing کشف شده اند و مهاجمان قصد دارند تا شکل کلی تر حملات لبریزی بافر را کشف نمایند (همانطور که در بخش ۲ شرح داده شد).

point Guard -2 : یک نسخه تصمیم یافته از روش stack Guard است که برای این پدیده طراحی شده است. point Guard دفاع stack Guard را تعمیم می دهد تا Canary ها را در مجاورت همه مکان نمای کد (مکان نماهای تابع و بافرهای long jmp) قرار دهد و اعتبار این Canary ها را کنترل کند و هر زمانی که یک مکان نمای کد مرجع زدایی شود. اگر Canary آسیب دیده باشد، آنگاه مکان نمای کد code pointer خراب می شود و برنامه باید یک هشدار تهاجم را صادر نماید. دو موضوع در رابطه با تجهیز مکان نماهای کد با حفاظت Canary مطرح است :

تخصیص Canary : فضا برای کلمه Canary که باید تخصیص داده شود وقتی که متغیر مورد حفاظت قرار گیرند، تخصیص داده می شود. و Canary باید مقدار دهی

اولیه شود هنگامی که متغیر مقداردهی می گردد. این موضوع مشکل آفرین است : برای حفظ سازگاری با برنامه های موجود ما نمی خواهیم اندازه متغیر حفاظت شده را تغییر دهیم بنابراین نمی توانیم کلمه Canary را به تعریف ساختار دیتا اضافه کنیم. بلکه تخصیص فضا برای کلمه Canary یعنی نواحی اطلاعات استاتیک مستقل در مقابل ساختارها و آرایشی ها باید در نظر گرفته شود.

کنترل کردن Canary یکپارچگی متغیر Canary لازم است تایید شود هر بار که متغیر حفاظت شده از حافظه بداخل یک رجیستر دور می شود یا خوانده می شود. این نیز مشکل آفرین است زیرا عمل خواندن reading از حافظه بخوبی در ترکیب معانی کامپایلر بخوبی تعریف نمی شود و کامپایلر هنگامی بیشتر مورد توجه است که متغیر واقعا بکار برود و الگوریتم های بهینه سازی مختلف بتواند متغیر از حافظه را بداخل رجیستر ها دور نماید هر زمان که احتیاج باشد. عملیات لازم است برای تمام شرایطی انجام شود که باعث می شود تا مقدار از حافظه خوانده شود. ما یک نسخه اولیه از point Guard را تهیه کرده ایم که حفاظت Canary را برای مکان نماهای تابع فراهم می کند که بطور استاتیک تخصیص داده می شوند (یعنی یک struct یا یک آرایه) این اجرا از کامل بودن فاصله دارد. وقتی که point Guard کامل باشد، ترکیب stack Guard و point Guard با برنامه های قابل اجرایی تولید که نسبت به حملات

لبریزی بافر بسیار ایمنی می باشند. تنها شکل حمله لبریزی بافرکه یک متغیر غیر مکان نما را مورد آسیب قرار میدهد و بر روی منطق برنامه تاثیر می گذارد ومورد توجه point Guard نمی باشد. برای ذکر این مشکل، کامپایلر point Guard شامل یک کلاس ذخیره Canary خاص است که حفاظت Canary را بر روی متغیرهای اختیاری اعمال می کند. بنابراین برنامه نویس می تواند بطور دستی حفاظت point Guard را به هر متغیری اضافه کند که بنظر می آید حساس به امنیت است.

3.5 ملاحظات عملکرد و سازگاری : کنترل کردن یکپارچگی مکان نمای کد نسبت به کنترل کردن حدود دارای معایبی است که بطور کامل مشکل لبریزی بافر را برطرف نمی کند، با اینحال دارای مزایای اساسی برحسب عملکرد، سازگاری با کد موجود و اجرا می باشد.

عملکرد : کنترل کردن حدود باید یک کنترل را در هر باری انجام دهد که یک عنصر آرایه خوانده یا نوشته می شود. بر عکس، کنترل کردن یکپارچگی مکانم نمای کد باید یک کنترل را انجام دهد هر بار که یک مکان نمای کد مرجع زدایی می شود یعنی هر بار که یک تابع برگردانده می شود یا یک مکان نمای تابع احضار می گردد. در کد C، مرجع زدایی مکان نمای کد کمتر از مراجع آرایه رخ می دهد و بالاسری پایینی ضروری را اعمال می کند. حتی code C^{++} ، در جایی که روش های مجازی فراخوانی های

تابع غیر مستقیم متداول است، ممکن است به آرایه ها بیشتر از فراخوانی روش های مجازی دسترسی داشته باشد که این امر بستگی به برنامه کاربردی دارد.

تلاش اجرایی: مشکل اصلی همراه با کنترل حدود برای کد C آن است که ترکیب معانی C امکان تعیین کردن حدود یک آرایه را دشوار می کند. C مفهوم یک آرایه را با مفهوم یک مکان نمای ژنریک برای بررسی شئ مخلوط می کند طوری که مراجعه به یک آرایه از عناصر از نوع FOO از یک مراجعه مکان نما به یک شئ Object از نوع foo ، غیر قابل تمایز است. چون یک مکان نما برای یک شئ مجزا معمولا فاقد حدود مرتبط با آن می باشد، هیچ جایی برای ذخیره کردن اطلاعات حدود وجود ندارد. بنابراین اجراهای کنترل حدود برای C لازم است تا برای روش های exotic خارجی مرتب شود تا اطلاعات حدود بازیابی گردد و مراجع آرایه مکان نماهای ساده ای نمی باشند بلکه مکان نماهایی برای شرح دهنده های بافر می گردند.

سازگاری با کد موجود : بعضی از روش های کنترل کردن حدود از قبیل Jones و Kelley قصد دارند تا سازگاری را با برنامه های موجود حفظ کنند و کنترل حدود را انجام دهند بطوری که« site of (in) = = site of (void*)» را تحت تاثیر قرار می دهند. سایر اجراها برای تبدیل یک مکان نما به صورت سه تایی «پایه و حدود » ، «جاری و پایان» یا تغییری از آن می باشد. این امر قرار دادن C معمولی site of (in) =

site of = » را نقص می کند و یک کامپایلر از نوع C را تولید می کند که می تواند یک زیر مجموعه محدود از برنامه های C را کامپایل کند بویژه آنهایی که از مکان نماها استفاده نمی کنند یا آنهایی که با چنین کامپایلری سرکار دارند، بسیاری از ادعاهای مزایای کنترل یکپارچگی مکان نمای کد در مقابل کنترل کردن حدود، نظری می باشند (ثابت نشده اند). با اینحال این امر بدلیل فقدان برجسته یک کامپایلر کنترل کننده حدود موثر برای کد C است. هیچ کامپایلر کنترل کننده حدودی وجود ندارد که قادر به تامین توانایی های سازگاری و عملکرد کامپایلر stack Guard باشد. این امر ادعاهای سازگاری و سهولت اجرا را پشتیبانی می کنند، برای آزمایش ادعاهای عملکرد ما، کسی باید تلاش برای ایجاد تقویت کنترل کننده حدود سازگار کامل را برای یک کامپایلر C سرمایه گذاری کند که برخلاف purify برای اشکال زدایی در نظر گرفته نمی شود.

4- ترکیبات موثر : در اینجا ما انواع آسیب پذیری ها و حملات مشروح در بخش ۲ را با اقدامات دفاعی مشروح در بخش ۳ مقایسه می کنیم تا تعیین کنیم که چه ترکیباتی از روش ها توانایی برای حذف کامل مشکل لبریزی بافر را حذف می کنند و با چه هزینه ای.

جدول ۳ حملات لبریزی بافر و دفاع ها را نشان می دهد. در قسمت بالا مجموعه ای از محل هایی دیده می شود در جایی که کد حمله قرار داده می شود (بخش 2.1) و در

قسمت پایین مجموعه روش هایی برای خراب کردن جریان کنترل برنامه (بخش 2.2) قرار دارد. در هر سلول، مجموعه اقدامات دفاعی قرار دارد که در مقابل آن ترکیب خاص موثر است. ما دفاع کنترل کردن حدود (بخش 3.3)از جدول ۳ را نادیده می گیریم. در حالیکه کنترل کردن حدود در جلوگیری کردن از تمام شکل های حمله لبریزی بافر موثر است، هزینه ها نیز در بسیاری از موارد، مانع می باشند.

متداولترین شکل حمله لبریزی بافر، عبارتند از حمله در مقابل یک رکورد فعال سازی است که کد را بداخل یک بافر تخصیص یافته به stack تزریق می کند. این شکل را از مقالات منتشر شده در اواخر ۱۹۹۶ می توانید پیدا کنید. عجیب نیست که روش دفاع های اولیه و stack Guard هر دو در مقابل این سلول موثر هستند. Stack غیر قابل اجرا ستون column را برای پوشش cover تمام حملاتی توسعه می دهد که کد را بداخل بافر های تخصیص یافته به stack تزریق می کند و دفاع stack Guard توسعه می یابد تا تمام حملات ای را شامل شود که رکوردهای فعال سازی را خراب می کند. این دفاع ها با یکدیگر بطور کامل سازگار هستند و بنابراین استفاده از both پوشش ضروری میدان حملات احتمالی را فراهم می کند. از حملات باقیمانده ای که توسط ترکیب stack غیر قابل اجرا پوشش نمی یابد و از دفاع stack Guard ، بسیاری از موارد می توانند بطور خودکار توسط کنترل یکپارچگی مکان نمای کد پیشنهاد شده

توسط point Guard ، جلوگیری شوند. حملات باقیمانده که متغیرهای برنامه اختیاری را خراب می کنند می توانند توسط point Guard بطور اسمی ذکر شوند، ولی مستلزم مداخله دستی چشمگیری می باشند. دفاع point Guard کاملا خودکار مستلزم کنترل کردن یکپارچگی Canary بر روی تمام متغیرها است در نقطه ای که کنترل کردن حدود شروع می شود تا با کنترل کردن یکپارچگی رقابت نماید.

جالب است توجه کنیم که در اولین حمله لبریزی بافر بکار رفته برای این دسته اخیر لبریزی بافر برای خراب کردن یک نام فایل است و با اینحال حملات لبریزی بافری از این روش استفاده نمی کند، علیرغم این حقیقت که هیچکدام از روش های دفاعی پیشنهاد شده در مقابل شکل از حمله موثر نمی باشد. معلوم نیست که آیا حملات لبریزی بافر بر پایه منطق بدلیل آن است که چنین آسیب پذیری هایی بسیار غیر معمول هستند یا بدلیل آن است حملات آسان تر انجام می شوند وقتی که مکان نماهای کد لحاظ شوند.

5- نتیجه گیری ها : ما یک تحلیل جامعی از آسیب پذیری های لبریزی بافر، حملات و دفاع ها را انجام دادیم، لبریزی های بافر ارزش این میزان تحلیل را دارند زیرا آنها اکثریت موضوعات آسیب پذیری امنیتی را بوجود می آورند نتایج این تحلیل نشان می دهند که ترکیبی از دفاع stack Guard و دفاع stack غیر قابل اجرا باعث شکست

دادن بسیاری از حملات لبریزی بافر معاصر شده است و دفـاع point Guard پیشـنهاد شده اکثریت حملات لبریزی بافر معاصر باقیمانده را ذکر می کند. شکل خاصی از حمله لبریزی بافر بکار رفته توسط Morris در ۱۹۹۸ برای popularte مورد قبول مردم قرار دادن روش لبریزی بافر، در حملات معاصر غیر قابل قبول است و به آسانی با استفاده از روش های موجود دفاع نمی شود.

جدول 3 : دفاع و حمله لبریزی بافر