دانشگاه صنعتی شریف

دانشکده‌ی مهندسی کامپیوتر

پایان‌نامه‌ی کارشناسی

گرایش مهندسی نرم‌افزار

عنوان:

تزریق آسیب‌پذیری در سطح کد منبع

نگارش:

سید پارسا طایفه مرسل

استاد راهنما:

دکتر مهدی خرازی

شهریور ۱۳۹۸


بسم الله الرحمن الرحیم

به نام خدا

دانشگاه صنعتی شریف

دانشکده‌ی مهندسی کامپیوتر

پایان‌نامه‌ی کارشناسی

عنوان: تزریق آسیب‌پذیری در سطح کد منبع

نگارش: سید پارسا طایفه مرسل

کمیته‌ی ممتحنین

استاد راهنما: دکتر مهدی خرازی امضاء:

استاد ممتحن: دکتر امیرعلی آبام امضاء:

تاریخ:

چکیده

امروزه یکی از چالش‌های اصلی متخصصین حوزه‌ی امنیت، سنجش دقت و درستی ابزارهای موجود برای کشف آسیب‌پذیری در برنامه‌هاست که عدم وجود پیکره‌های معیار، ارزیابی این ابزارها را بسیار دشوار کرده است. از این رو تزریق[۱] آسیب‌پذیری به کد منبع[۲] جهت ایجاد پیکره‌های[۳] معیار به صورت خودکار و با برچسب‌گذاری دقیق(شامل نوع آسیب‌پذیری، محل آن و رویه‌ی فعال‌سازی آن)، کمک شایانی به امر ارزیابی ابزارهای امنیتی و به طور کلی ایمن‌سازی نرم‌افزارها می‌کند. از این رو در این پژوهش به طراحی و توسعه‌ی ابزاری با هدف تزریق آسیب‌پذیری در سطح کد منبع می‌پردازیم. ابزار ارائه شده با تزریق و فعال‌سازی آسیب‌پذیری‌های جدا شده از کدهای آسیب‌پذیر شناخته شده، ضمن داشتن برتری نسبت به ابزارهای موجود از نظر واقعی و غیر ساختگی بودن و عدم محدودیت در نوع آسیب‌پذیری‌های تولید شده، بستر مناسبی را برای تولید پیکره‌های معیار در سطح انبوه فراهم می‌کند.

کلیدواژه‌ها: کشف آسیب‌پذیری، تزریق آسیب‌پذیری، کد منبع، پیکره‌های معیار

---

۱ Injection
۲ Source Code
۳ Corpus

فهرست مطالب

فهرست شکل‌ها

فصل ۱

مقدمه

۱ـ۱ تعریف صورت مسئله

ابزارهای کشف آسیب‌پذیری یکی از مهمترین عناصر در دنیای ایمنی و امنیت نرم‌افزارها به حساب می‌آیند به طوری که کیفیت و عملکرد آن‌ها با امنیت در نرم‌افزارها و سیستم‌ها ارتباط مشهود و تنگاتنگی دارد. بدیهی است که بهبود دقت و عملکرد این ابزارها باعث کشف آسیب‌پذیری‌های بیشتر و در نتیجه کاهش خسارات ناشی از حملات و نقض‌های سیستمی می‌گردد.

یکی از اصلی‌ترین مشکلاتی که توسعه‌دهندگان این ابزارها با آن روبه‌رو هستند، کمبود و فقدان پیکره‌های معیار جهت سنجش دقت و کیفیت این ابزارها می‌باشد. از آن جایی که در فرایند توسعه‌ی این ابزارها از آسیب‌پذیری‌های موجود و شناخته شده استفاده شده است، سنجش این ابزارها با همان آسیب‌پذیری‌ها نتایج معنی‌داری را ارائه نمی‌دهد و عملکرد رضایت‌بخش آن‌ها، بر روی این آسیب‌پذیری‌ها لزوماً معیار معتبری برای سنجش عملکرد آن‌ها در مواجهه با آسیب‌پذیری‌های جدید نمی‌باشد.

تولید پیکره‌های معیار جهت کیفیت‌سنجی ابزارهای کشف آسیب‌پذیری گامی مهم در پاسخ به مشکل فوق می‌باشد. این پیکره‌ها که شامل تعداد بسیار زیادی کد منبعِ آلوده به انواع آسیب‌پذیری‌ها در عمق و سطوح مختلف می‌باشند، می‌توانند به خوبی توانایی این ابزارها را به چالش کشیده و ارزیابی دقیقی از عملکر آن‌ها نسبت به نوع و پیچیدگی آسیب‌پذیری‌ها ارائه دهند. به طوری که می‌توان هر ابزار را در مقابل طیف وسیعی از آسیب‌پذیری‌ها مورد آزمایش، و کیفیت و عملکرد آن را در هر دسته به طور

مجزا مورد بررسی قرار داد.

به دلیل محدودیت در تعداد آسیب‌پذیری‌های کشف شده تا به امروز، همچنین هزینه‌ی بالای کشف آسیب‌پذیری‌های جدید به صورت دستی، تولید این پیکره‌ها به چالشی برای محققین این حوزه تبدیل شده است.

یکی از روش‌های پاسخ به این مشکل، آلوده‌سازی کدهای منبعِ موجود می‌باشد. به این صورت که آسیب‌پذیری به گونه‌ای در یک کد تمیز جاسازی و یا ایجاد می‌شود و در نتیجه‌ی آن، کد حاصل به یک یا چند آسیب‌پذیری از نوع خاص و در عمق دلخواه، آلوده می‌گردد.

با خودکارسازی فرایند فوق می‌توان به راحتی تعداد بسیار زیادی کد منبع از نرم‌افزارهای متن‌باز را به انواع آسیب‌پذیری‌ها به صورت برچسب‌گذاری شده، آلوده ساخت و پیکره‌های معیار متعددی برای سنجش ابزارهای کشف آسیب‌پذیری تولید کرد.

۱ـ۲ اهمیت موضوع

تولید پیکره‌های معیار در تعداد بالا و با پیچیدگی‌های گوناگون، امری حیاتی در توسعه ابزارهای کشف آسیب‌پذیری و در نتیجه‌ی آن، بهبود امنیت در سطح تمام سیستم‌ها و نرم‌افزارها می‌باشد.

یکی از مهمترین رویدادها در دنیای امنیت سیستم‌ها و اطلاعات، مسابقات فتح پرچم[۱] می‌باشد. این مسابقات با فراهم کردن محیطی رقابتی و حرفه‌ای، محل شکوفایی بسیاری از متخصصین حوزه‌ی امنیت، و علاوه بر آن باعث ایجاد انگیزه و آگاهی برای ورود به دنیای امنیت می‌باشند. از اصلی‌ترین چالش‌هایی که برگزارکنندگان این مسابقات با آن رو به رو هستند، پیچیدگی و هزینه‌ی بالای طرح سوالات مناسب برای هر مسابقه می‌باشد. ابزارهای تزریق آسیب‌پذیری می‌توانند در تسهیل امر تولید سوالات گوناگون، در دسته بندی‌های دلخواه و با پیچیدگی متفاوت، تاثیر بسزایی داشته باشند.

[۱] Capture The Flag (CTF)

فصل ۲

مفاهیم اولیه

۲ـ۱ آسیب‌پذیری

به هر نقصی که بتواند باعث ایجاد خطر برای تمامیت، محرمانگی و یا دسترسی‌پذیری در کل یا بخشی از سیستم شود، آسیب‌پذیری[۱] گفته می‌شود.

آسیب‌پذیری‌ها می‌توانند در سطوح گوناگون سیستم از جمله نرم‌افزارها، سخت‌افزارها، پروتکل‌ها ارتباطی، الگوریتم‌ها و ... به دلایل مختلف وجود داشته باشند، که از مهمترین آن‌ها می‌توان به اشتباهات برنامه‌نویسی، ایرادات سخت‌افزاری و بی‌توجهی به معماری‌های امن در طراحی سیستم اشاره کرد. وجود آسیب‌پذیری در هر سطح از یک سیستم، می‌تواند تهدیدی برای امنیت تمام سطوح بالاتر سیستم باشد. به طور مثال، وجود یک ایراد در طراحی واحد مدیریت حافظه[۲] در واحد پردازنده‌ی[۳] سیستم(سطح بسیار پایین و نزدیک به سخت‌افزار)، می‌تواند پس از بهره‌وری[۴] توسط یک یا چند عامل متخاصم، باعث افشای ایمیل‌های یک سازمان نظامی(سطح بسیار بالا و نزدیک به کاربر) شود.

هر ساله آسیب‌پذیری‌ها، هزینه و خسارات بسیار سنگینی به سازمان و شرکت‌ها تحمیل می‌کنند که بخش اعظمی از آن صرف جبران صدمات وارده در حملات سایبری و کشف آسیب‌پذیری‌های پنهان می‌شود. همین امر باعث بالا بودن قیمت آسیب‌پذیری‌های جدید و گزارش نشده می‌باشد که در بازارهای

[۱] Vulnerability
[۲] Memory Management Unit(MMU)
[۳] Central Processing Unit(CPU)
[۴] Exploit

غیر قانونی و زیر زمینی مانند وب عمیق[۵]، قالباً به منظور سود جویی معامله می‌گردند.

شرکت‌ها و سازمان‌های مختلف، برای کشف کم هزینه‌تر آسیب‌پذیری‌های موجود در زیرساخت‌ها و برنامه‌های خود، بعضاً اقدام به برگزاری مسابقات شکار نقص[۶] کرده و به ازای هر آسیب‌پذیری کشف شده، مبلغی را با متناسب با جدیت و خطر آسب‌پذیری کشف شده، به فرد مذکور جایزه می‌دهند.

۲ـ۲ ابزارهای کشف آسیب‌پذیری

به مجموعه‌ای از ابزارها که به شیوه‌های گوناگون و در سطوح مختلف یک برنامه یا سیستم، اقدام به کشف آسیب‌پذیری‌ها[۷] می‌کنند، گفته می‌شود. این ابزارها به دو دسته‌ی اصلی پویا[۸] و ایستا[۹] تقسیم می‌شوند. ابزارهای مبتنی بر تحلیل پویا با نظارت بر اجرای و زیر نظر گرفتن رفتار موئلفه‌های مختلف برنامه، به دنبال نقص‌ها و رفتارهای غیر عادی می‌گردنند. این در حالیست که ابزارهای مبتنی بر تحلیل ایستا بدون نیاز به اجرای برنامه و تنها با بررسی کد منبع، کد دودویی[۱۰] و یا نمایش میانی[۱۱] اقدام به بررسی اجزای تشکیل‌دهنده‌ی برنامه می‌کنند.

بدیهی است که هیچ یک از دو تحلیل فوق، برتری قاطعی نسبت به دیگری نداشته و هر یک نقاط ضعف و قوت مخصوص به خود را دارند.

۲ـ۳ تزریق آسیب‌پذیری

به عمل آلوده ساختن یک کد تمیز(فاقد آسیب‌پذیری) به یک یا چند آسیب‌پذیری، تزریق یا جاسازی آسیب‌پذیری گویند. این عمل به شیوه‌های گوناگون از جمله افزودن کد آلوده، پاک کردن تدابیر امنیتی اتخاذ شده در کد و یا ایجاد تغیراتی کوچک در کد هدف، انجام می‌گردد.

نکته‌ی اساسی در تزریق آسیب‌پذیری این است که کد آلوده شده باید همچنان قابل اجرا بوده و

[۵] Deep Web
[۶] Bug Bounty
[۷] Vulnerability Discovery
[۸] Dynamic
[۹] Static
[۱۰] Binary
[۱۱] Intermediate Representation(IR)

تمامی خصوصیات رفتاری کد تمیز را داشته و صرفا یک یا چند رفتار آسیب‌پذیر به مجموعه رفتارهای آن افزوده شده باشد. بدیهی است تزریق آسیب‌پذیری به صورتی که کد نهایی غیرقابل اجرا و یا تبدیل به ماهیت رفتاری متفاوتی شده است، فاقد ارزش فنی می‌باشد.

همچنین لازم به ذکر است که آسیب‌پذیری‌های تزریق شده نباید بدیهی[۱۲] باشند. به بیان دقیق‌تر، نباید در تمام اجراهای برنامه فعال شده و یا نمایان گردند. بلکه باید همانند آسیب‌پذیری‌های واقعی، تحت شرایط خاص و به ازای مقادیر خاصی از ورودی‌ها فعال شوند. ارائه‌ی مقادیر فعال‌ساز آسیب‌پذیری‌های تزریق شده امری ضروری نبوده اما امتیاز مثبتی در ارزیابی ابزار تولید آسیب‌پذیری به حساب می‌آید.

[۱۲] Trivial

فصل ۳

بررسی راهکارهای پیشین

در پاسخ به مسئله‌ی مطرح شده، گروه‌های متعددی از متخصیص اقدام به تحقیق و توسعه‌ی ابزارهای تزریق آسیب‌پذیری جهت تولید خودکار پیکره‌های معیار کرده‌اند که در این بخش به اختصار به بررسی دو مورد خواهیم پرداخت.

۳ـ۱ لاوا

لاوا[۱] [۱] یکی از مطرح‌ترین ابزارهای خودکار تزریق و آلوده‌سازی کد منبع می‌باشد. این ابزار با تحلیل کد منبع نقاطی را به عنوان نقطه‌ی حمله انتخاب کرده، سپس با سوء استفاده از وابستگی آن خط به مقدار یک متغیر محلی، اقدام به آسیب‌پذیر ساختن کد به وسیله‌ی رساندن مقدار متغیر حساس، به مقداری خاص می‌کند.

انتخاب نقطه‌ی حمله[۲] بر اساس دسترسی‌پذیری آن خط از نقطه‌ی شروع برنامه به ازای فضای حالت ورودی انجام می‌پذیرد. همچنین متغیر استفاده شده برای حمله، بر اساس فاصله‌ی پیچیدگی محاسباتی آن متغیر در نقطه‌ی حمله با متغیر ورودی، انتخاب می‌شود. تکه کد زیر نمونه‌ای از عملکر این ابزار را نشان می‌دهد.

---

[۱] LAVA

[۲] Attack Point

```
1  void foo(int a, int b, char *s, char *d, int n){
2      int c = a + b;
3      if(a!=0xdeadbeef)
4          return;
5      for(int i=0;i<n;i++)
6          c+=s[i];
7      memcpy(d,s,n+c) //Original Source
8      // BUG: memcpy(d+ (b==0x6c617464)*b,s,n+c);
9  }
```

شکل ۳ـ۱: نمونه‌ای از عمکرد لاوا بر روی یک کد منبع

همانطور که در تکه کد فوق مشخص شده است، خط شامل تابع memcpy که از توابع مشهور آسیب‌پذیر در زبان سی بشمار می‌رود، به عنوان نقطه‌ی حمله انتخاب شده است. با استفاده از ارتباط مقدار متغیر b با ورودی تابع سطح بالاتر و در دسترس‌بودن آن در اطراف نقطه‌ی حمله، این متغیر جهت فعال‌سازی[۳] حمله مورد استفاده واقع شده است. به طوری که در عبارت جایگزین شده، اگر مقدار متغیر b برابر با یک مقدار جادویی [۴] شود، مولفه‌ی ورودی اول تابع memcpy مقدار بسیار بزرگی شده و این باعث رخ دادن آسیب‌پذیری سرریز بافر[۵] در برنامه می‌شود.

از نقاط قوت این ابزار می‌توان به انتخاب هوشمندانه‌ی نقطه و متغیر حمله اشاره کرد. پارامترهای در نظر گرفته شده در این انتخاب، باعث ایجاد آسیب‌پذیری تزریق شده در قسمت عمیقی از برنامه شده که این امر باعث به چالش کشیده شدن ابزارهای کشف آسیب‌پذیری می‌گردد. این درحالیست که وابستگیِ فعال شدن حمله به یک مقدار جادوییِ تصادفی، کشف آسیب‌پذیری‌ها را برای بعضی ابزارها، به امری تقریباً غیر ممکن و دور از واقعیت تبدیل ساخته است.

[۳] Trigger
[۴] Magic Value
[۵] Buffer Overflow

۳ـ۲ اویل کدُر

اویل کدر[۶] [۲] که یکی دیگر از ابزارهای خودکار آلوده‌سازی کد منبع می‌باشد، با حذف شروط امنیتی در کد، اقدامات پیشگیرانه‌ی توسعه‌دهنده‌ی برنامه را منحل کرده و باعث آسیب‌پذیر شدن کد منبع، دقیقا در همان مکان‌هایی که توسعه‌دهنده قصد جلوگیری از آن را داشته می‌شود.

به طور دقیق‌تر، اویل کدر با بررسی منبع‌ها[۷] و چاه‌های[۸] داده و تحلیل جریان کنترلی[۹] و جریان داده‌ای[۱۰] بین این دو، اقدام به کشف مسیرها و نقاط حساسی که توسط شروع امنیتی حفاظت شده‌اند می‌کند. تکه کد زیر شیوه‌ی نگرش این ابزار را روشن‌سازی می‌کند.

```
1  void copy_buffer(File *f_true, File *f_false, char *buf, int which_file)
2  {
3      int len;
4      if (which_file)
5          len = read_from_file(f_true);
6      else
7          len = read_from_file(f_false);
8
9      if (len > 256)
10     {
11         printf("ERROR: len is too big");
12         exit(1);
13     }
14     char local[256];
15     memcpy(local, buf, len);
16     memset(buf, 0, 512);
17     do_something_with(local);
18 }
```

شکل ۳ـ۲: نمونه‌ای از عملکرد اویل کدر بر روی یک کد منبع

کد فوق یک تابع بسیار امن است که هدف آن کپی کردن مقدار یک فایل در یک بافر محلی می‌باشد. برنامه‌نویس در این کد، با آگاهی از خطرات استفاده از تابع `memcpy`، به طور هوشمندانه‌ای شرط امنیتی `if(len > 256)` را قبل از اجرای این تابع قرار داده است تا مانع شکل‌گیری آسیب‌پذیری سرریز بافر

[۶] EvilCoder
[۷] Source
[۸] Sink
[۹] Control Flow
[۱۰] Data Flow

شود. اویل کدر با شناسایی تابع memcpy به عنوان یک چاه داده‌ای ارزشمند، و با شناسایی if به عنوان یک راس[۱۱] در تعیین جریان داده‌ای و کنترلی، اقدام به حذف این if و در نتیجه‌ی آن، آسیب‌پذیر ساختن کد در خط فراخوانی تابع memcpy می‌کند.

از نقاط ضعف این ابزار میتوان به محدود بودن آسیب‌پذیری‌های قابل ساخت با این روش و امکان تغییر ماهیت برنامه با حذف خطوط تعیین‌کننده‌ی جریان داده‌ای و کنترلی اشاره کرد.

۳ـ۳ باگ سنتسایز

یک دیگر از ابزارهای جدید برای آلوده‌سازی کد در سطح کد منبع، باگ سنتسایز[۱۲] [۳] می‌باشد. این ابزار با تزریق یک ماشین خودکار[۱۳] در درون کد و با وابسته‌سازی انتقال[۱۴] بین وضعیت‌های[۱۵] آن ماشین و متغیرهای برنامه‌ی هدف، اقدام به اجرای یک برنامه‌ی کوچک در دل برنامه‌ی هدف، به صورت موازی و وابسته به آن می‌کند. اگر ماشین خودکار تزریق شده، که در واقع یک ماشین حالت محدود تصمیم‌پذیر[۱۶] است، به وضعیت نهایی خود برسد، با استفاده از فراخوانی assert(false) اقدام به منحل کردن اجرای برنامه می‌کند که این امر در واقع شبیه‌سازی فعال شدن یک آسیب‌پذیری می‌باشد.

به طور دقیق‌تر، این ابزار با استفاده از یکی از ابزارهای معروف اجرای نمادین[۱۷] به نام KLEE اقدام به کشف مقادیر متغیرها در طول یک اجرا به ازای ورودی‌های مختلف می‌کند. سپس با تحلیل نتایج حاصل از اجرای نمادین و با داشتن مقدار هر متغیر در هر مکان از کد در حین اجرا، اقدام به تولید عبارات شرطی لازم جهت تعیین وضعیت ماشین خودکار می‌کند.

در تولید این عبارات شرطی به این نکته توجه شده است که شرط‌های تولید شده بدیهی نبوده و فقط در طیف کوچکی از اجراها باعث تغیر وضعیت ماشین خودکار به وضعیت بعدی شوند. این امر کشف آسیب‌پذیری تولید شده را تا حدی دشوار می‌کند.

دو شکل زیر به ترتیب یک کد آلوده شده با این ابزار و ماشین خودکار تزریق شده را نشان می‌دهد.

---

[۱۱] Node
[۱۲] Bug Synthesis
[۱۳] Automaton
[۱۴] Transition
[۱۵] State
[۱۶] Deterministic Finite Automaton
[۱۷] Symbolic Execution

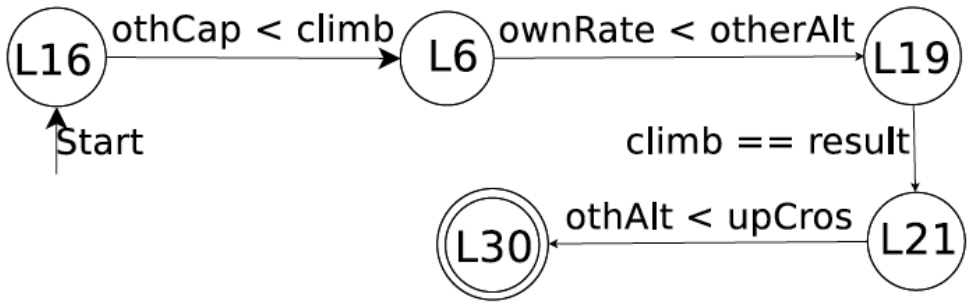
```
void ALIM ()
{
 othCap = climb + altVal;
 L6: /* if (ownRate < otherAlt && state == 6)
          state = 19; */
}

int InhibitBiasedClimb ()
{
 int up,down;
 up = upSep + 100 +altVal;
 down = upSep + OtherTrackedAlt;
 L16: /* if (othCap < climb && state == 16)
           state = 6;*/
 return (climb ? up : down);
}

void main()
{
 input(curSep, ownAlt, ownRate, otherAlt,
       altVal, upSep, downSep, othCap, climb);
L0:
 /* state = 16; */
 upPref = InhibitBiasedClimb() + downSep;
 upCross = ownAlt + otherAlt;
 ownRate = ownRate + curSep;
 ALIM();
 if (uppref > 5500){
   result = altVal;
 L19: /* if (climb == result && state == 19)
           state = 21;*/
  }
 L21: /* if (othAlt < upCros && state == 21)
           state = 30;*/
 upCross = ownAlt - otherAlt;
 L30: /*if (state == 30) assert(0);*/
}
```

شکل ۳ـ۳: نمونه‌ای از عملکرد باگ سنتسایز بر روی یک کد منبع

برای نمایش بهتر، وضعیت‌های ماشین خودکار به صورت برچسب کنار خطوط مربوطه مشخص شده‌اند. با ارضای شرط در برگیرنده، وضعیت ماشین که با یک مقدار عدد صحیح ذخیره شده است، تغیر می‌کند. به محض رسیدن ماشین خودکار به وضعیت پایانی، برنامه سقوط[۱۸] می‌کند.

در شکل زیر ماشین خودکار تزریق شده به صورت بریده شده از کد قابل مشاهده است.

[۱۸] Crash



شکل ۳ـ۴: ماشین خودکار تزریق شده و وابستگی آن به وضعیت متغیرهای محلی برنامه‌ی هدف

یکی از نقاط ضعف مشهود این ابزار، امکان جدا سازی ماشین خودکار تزریق شده به وسیله‌ی برشگرهای برنامه می‌باشد. از این رو کشف آسیب‌پذیری‌های تولید شده توسط این ابزار برای ابزارهای کشف آسیب‌پذیری مبتنی بر برش برنامه، امری ساده به حساب می‌آید.

۳ـ۴ سی‌تی‌اف خودکار

سی‌تی‌اف خودکار[۱۹] [۴] یک مسابقه‌ی فتح پرچم می‌باشد که سوالات آن به وسیله‌ی ابزار معرفی شده در بخش قبل، لاوا، ایجاد شده‌اند. همانطور که در بخش اهمیت موضوع ذکر شد، یکی از چالش‌های اصلی برگزارکنندگان این مسابقات، هزینه‌ی بالای طرح سوالات جدید می‌باشد. از این رو تولید سوالات به وسیله‌ی ابزارهای آلوده‌سازی کد می‌تواند کمک شایانی به تسهیل برگزاری این مسابقات کند.

با بررسی نتایج مسابقه فتح پرچم برگزار شده و تحلیل نظرات شرکت کنندگان می‌توان دریافت که سوالات تولید شده توسط لاوا علی رغم نداشتن تنوع مطلوب، چالش خوبی هم برای شرکت‌کنندگان کهنه‌کار و هم شرکت‌کنندگان تازه‌کار بوده‌اند.

سی‌تی‌اف خودکار با نشان دادن پتانسیل چنین رویدادهایی، بر اهمیت توسعه ابزارهای تولید خودکار آسیب‌پذیری تاکید می‌کند.

AutoCTF[۱۹]

۳ـ۵ نقاط ضعف مشترک ابزارهای موجود

با بررسی دقیق ابزارهای معرفی شده در فوق و دیگر ابزارهای موجود در این حوزه، می‌توان به نقاط ضعف مشترکی در طراحی و عملکرد آن‌ها پی برد.

- استفاده از دانش خبره[۲۰]

یکی از اصلی‌ترین ایرادات وارده به ابزارهای فوق، استفاده از دانش خبره‌ی توسعه‌دهنده‌ی ابزار در طراحی الگوریتم تولید آسیب‌پذیری می‌باشد. به این صورت که طراح با در نظر گرفتن فاکتورهای خاصی و ارزش‌گزاری سلیقه‌ای آن‌ها، اقدام به رده‌بندی و انتخاب بین نقاط مختلف برنامه و یا نوع و شیوه‌ی عملکرد آسیب‌پذیری می‌کند. این نگرش نه تنها باعث ایجاد محدودیت در فضای آسیب‌پذیری‌های تولید شده می‌شود، بلکه باعث می‌شود تمام آسیب‌پذیری‌های تولید شده، رنگ و بوی خاصی به خود بگیرند که این با یکی از اصلی‌ترین اهداف تولید این ابزارها که تولید آسیب‌پذیری‌های متنوع از نظر ساختاری بود، در تناقض است.

همچنین باتوجه به تنوع ابزارهای کشف آسیب‌پذیری و روش‌های متفاوت آن‌ها، به کارگیری دانش خبره در تولید آسیب‌پذیری باعث موفقیت بیش از حد ابزارهای منطبق با آن نگرش و ضعف عملکرد ابزارهای دیگر می‌شود. به طور مثال، به دلیل وابستگی آسیب‌پذیری‌های تولید شده توسط لاوا به یک مقدار تصادفی، ابزارهای مبتنی بر روش فازینگ[۲۱]، عملکر بسیار مطلوبی داشته‌اند. این در حالیست که ابزارهای مبتنی بر برش برنامه، به دلیل ناتوانی در حدس مقدار تصادفی، نتوانستند موفقیتی در کشف این آسیب‌پذیری‌ها کسب کنند.

- محدودیت نوع آسیب‌پذیری‌ها

ابزارهای مذکور صرفاً قادر به تولید انواع بسیار محدودی از آسیب‌پذیری‌ها هستند. به طور مثال، هر دو ابزار فوق به جستجو توابع حساسی مانند `memcpy` یا `scanf` در کد، اقدام به ایجاد آسیب‌پذیری‌های سرریز بافر و یا فرمت رشته[۲۲]، با محوریت استفاده از این توابع می‌کنند. این درحالیست که این آسیب‌پذیری‌ها تنها بخش اندکی از انواع آسیب‌پذیری‌های شناخته شده بوده و آسیب‌پذیری‌های بسیار زیادی وجود دارند که صرفاً حول یک تابع خاص و شناخته شده تعریف نمی‌شوند.

[۲۰] Expert knowledge
[۲۱] Fuzzing
[۲۲] Format String

فصل ۴

راهکار پیشنهادی

۴ـ۱ کلیات روش

هر ساله تعداد زیادی آسیب‌پذیری در نرم‌افزارهای متن‌باز یافت شده و در قالب CVE[1] گزارش می‌شوند. همانطور که در بخش‌های قبل بیان شد، تعداد کل این کدها آنقدر نیست که به خودی خود بتوان از آن‌ها به عنوان پیکره‌های معیار استفاده کرد. اما نکته‌ی قابل تأمل اینجاست که این کدهای آلوده‌ی گزارش شده، هر یک دارای یک یا چند آسیب‌پذیری می‌باشند. حال اگر بتوان فقط قسمت آلوده‌ی کدها را به شکل یک تابع یا تکه کد آسیب‌پذیر جدا کرد، می‌توان آن‌ها را درون کدهای تمیز جدید تزریق کرده و بدین صورت، کدهای جدیدی که به یک آسیب‌پذیری کاملا واقعی آلوده می‌باشد، ایجاد کرد.

به بیان ساده‌تر، می‌توان هسته و شالوده‌ی آسیب‌پذیر را از دل یک برنامه‌ی آلوده که قبلا کشف و گزارش شده است، جدا کرده و به وسیله‌ی آن تعداد بیشماری کد را به راحتی به آن آسیب‌پذیری آلوده ساخت.

هر آسیب‌پذیری در کد فارغ از تاثیرات امنیتی، یک رفتار یا خاصیت به شمار می‌رود. یکی از اصلی‌ترین کاربردهای برشگرهای برنامه، جدا سازی یک خاصیت، تحت عنوان یک یا چند برش می‌باشد. از این رو، با دانستن مکان آسیب‌پذیری در یک کد آلوده، جدا سازی آن قسمت به کمک برشگرها امری کاملا ممکن و آزموده شده است که البته در حیطه‌ی کار این پروژه نمی‌باشد. پس با استفاده از این ابزارها

[1] Common Vulnerabilities and Exposures

و بررسی تمام مخازن[۲] کدهای آلوده‌ی متن‌باز، می‌توان به راحتی پیکره‌ی بزرگی از بدنه(تکه کد)هایی جمع آوری کرد که هر یک بالذات یک آسیب‌پذیری می‌باشند. با فرض داشتن چنین بدنه‌هایی، امکان آلوده کردن هر کدی در هر نقطه، با یک یا چند آسیب‌پذیری که واقعا توسط انسان به دلیل بی‌دقتی یا اشتباه برنامه‌نویسی رخ داده است، فراهم می‌شود.

عمل تزریق به دلیل عدم وابستگی بدنه‌ی آسیب‌پذیری به کد جدید، می‌تواند در هر سطح و عمقی از کد جدید صورت پذیرد. در هنگام اجرا، در صورت رسیدن مسیر اجرای برنامه به محل تزریق، آسیب‌پذیری فعال شده و تاثیر خود را بر روند اجرای برنامه می‌گذارد. بدیهی است که می‌توان انواع پیچیدگی‌ها و وابستگی‌ها را در کد تزریق شده نسبت به کد جدید ایجاد کرد تا کدهای آسیب‌پذیر تولید شده، تنوع بیشتری داشته باشند.

۴ـ۲ برتری نسبت به روش‌های پیشین

از آن جایی که ارزش یک پاسخ جدید به یک مسئله، به برتری آن پاسخ نسبت به پاسخ‌های موجود وابسته است، در این بخش به اختصار به برتری این روش در مقایسه با ایرادات مطرح شده در قسمت‌های قبل برای ابزارهای موجود می‌پردازیم.

- عدم بهره‌وری از دانش خبره

همانطور که در بخش‌های قبل بیان شد، یکی از اصلی‌ترین ایرادات ابزارهای موجود، استفاده از دانش خبره در تولید آسیب‌پذیری‌های جدید می‌باشد. این امر باعث محدودیت و شباهت زیاد بین آسیب‌پذیری‌های تولید شده، و در نتیجه، سنجش ناعادلانه‌ی ابزارهای کشف آسیب‌پذیری می‌شود. در این روش اما، از آنجایی که بدنه‌ی آسیب‌پذیر از نو تولید نشده است، هیچ امکانی برای ورود سلیقه‌ی شخصی طراح و استفاده از دانش خبره وجود ندارد.

در واقع این بدنه‌های آسیب‌پذیر، هر یک تکه کدی هستند که توسط یک توسعه‌دهنده نوشته شده و بعدا توسط عده‌ی دیگری کشف و گزارش شده‌اند. در نتیجه از منظر واقعی و غیرساختگی بودن، هیچ ایرادی بر آن‌ها وارد نیست. بدیهی است که ابزارهای کشف آسیب‌پذیری همواره در پی کشف آسیب‌پذیری در کدهای توسعه داده شده توسط فرد برنامه‌نویس هستند. از این رو نمی‌توان نمونه‌ی بهتری

[۲] Repositories

برای سنجش عملکر آن‌ها در جهت تشخیص آسیب‌پذیری‌های انسانی(نه ساخته‌ی ماشین)، نسبت به اشتباهاتی که قبلا توسط انسان رخ داده است، ارائه نمود.

- تنوع آسیب‌پذیری‌های تولید شده

از آنجایی که CVEهای موجود، شامل طیف بسیار گسترده‌ای از آسیب‌پذیری‌ها می‌باشند که در انواع مختلف و در سیستم‌های متفاوتی کشف شده‌اند، استفاده از آن‌ها برای آلوده‌سازی کدها باعث ایجاد کدهای آلوده با آسیب‌پذیری‌های بسیار متنوع می‌شود. در نتیجه برتری این روش نسبت به سایر روش‌ها که در آن‌ها تمرکز صرفاً بر روی چند تابع حساس با آسیب‌پذیری‌های تکراری بود، کاملا مشهود است.

فصل ۵

پیاده‌سازی

۵ـ۱ تولید گراف جریان کنترلی

گراف جریان کنترلی[۱] برای یک برنامه شامل بلوک‌های ابتدایی[۲] و شیوه‌ی ارتباط آن‌ها از نظر فراخوانی و پرش‌ها می‌باشد. به تعدادی دستور که به صورت متوالی و بدون پرش[۳] اجرا می‌شوند، یک بلوک ابتدایی گفته می‌شود. گراف جریان کنترلی باعث درک بیشتر از ساختار برنامه و روشن‌سازی عمق یک دستور از نظر میزان فراخوانی‌های تو در تو می‌شود. این امر در تصمیم‌گیری مبنی بر محل فعال‌سازی و فراخوانی آسیب‌پذیری از اهمیت بالایی برخوردار می‌باشد. در حقیقت هر چقدر آسیب‌پذیری در قسمت عمیق‌تر گراف جریان کنترلی جاسازی شود، دسترسی، فعال‌سازی و کشف آن دشوارتر می‌شود. زیرا برای رسیدن به آن خط از برنامه، پرش‌های شرطی و فراخوانی‌های متعددی، به ترتیبی خاص باید صورت پذیرند که این امر رسیدن ابزار کشف آسیب‌پذیری به آن نقطه را دشوارتر می‌کند. عکس مطلب بیان شده نیز صادق می‌باشد. تزریق آسیب‌پذیری در بلوک‌های ابتدایی نزدیک به راس اصلی گراف جریان کنترلی(شروع برنامه)، احتمال کشف آن را بالا برده و آسیب‌پذیری آسانی به حساب می‌آید.

لازم به تاکید است که از آنجایی که یکی از اهداف اصلی این روش عدم استفاده از دانش خبره در تولید پیکره‌ها می‌باشد، هیچ تلاشی مبنی بر پیچیده کردن آسیب‌پذیری‌های تزریق شده انجام نمی‌پذیرد. بلکه آسیب‌پذیری‌ها با استفاده از گراف جریان کنترلی، با هدف سنجش تمام ابزارهای کشف آسیب‌پذیری،

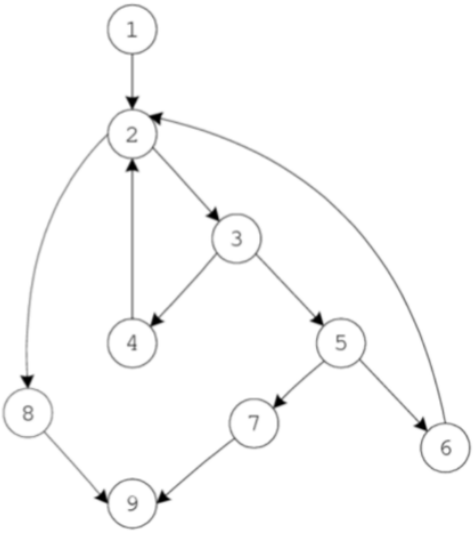
[۱] Control Flow Graph (CFG)
[۲] Basic Blocks
[۳] Jump

در عمق‌های مختلف و با پیچیدگی‌های مختلف به طور یکنواخت و تصادفی توزیع می‌گردند.

Source Program:

```
int binsearch(int x, int v[], int n)
{
    |  int low, high, mid;
  1 |  low = 0;
    |  high = n - 1;
    |  while (low <= high) |2
    |  {
    |     3 |  mid = (low + high)/2;
    |       |  if (x < v[mid])
    |       |        high = mid - 1;  |4
    |     5 |  else if (x > v[mid])
    |       |        low = mid + 1;   |6
    |     7 |  else return mid;
    |  }
    |  return -1; |8
} |9
```

CFG:



شکل ۵ـ۱: یک نمونه کد و گراف جریان کنترلی متناظر با آن

در شکل فوق بلوک‌های ابتدایی با اعداد نامگذاری شده‌اند و ارتباط بین آن‌ها به طور شماتیک نشان داده شده است.

در این پروژه برای تولید گراف جریان کنترلی از نرم‌افزار GCC استفاده شده است. این نرم‌افزار که یکی از قدرتمندترین و مطرح‌ترین کامپایلرها برای زبان سی و سی‌پلاس‌پلاس می‌باشد، امکانات زبانی و پردازشی زیادی را در اختیار کاربر قرار می‌دهد. از آنجایی که تولید گراف جریان کنترلی برای انجام امر کامپایل کردن برنامه ضروریست، GCC ضمن کامپایل برنامه، با دریافت یه پرچم[۴] خاص، گراف جریان کنترلی‌ای را که برای کامپایل برنامه تولید کرده است، در قالب یک فایل متنی در اختیار کاربر می‌گذارد. دستور به کار رفته برای تولید این گراف به شرح زیر می‌باشد.

```
1   gcc - fdump - tree - cfg - lineno
```

[۴] Flag

```
<bb 6>:
[2/httpd.c : 95:9] // predicted unlikely by continue predictor.
[2/httpd.c : 95:9] option = {CLOBBER};
goto <bb 10>;

<bb 7>:
[2/httpd.c : 96:17] D.4747 = [2/httpd.c : 96] p->ai_addrlen;
[2/httpd.c : 96:29] D.4748 = [2/httpd.c : 96] p->ai_addr;
[2/httpd.c : 96:17] listenfd.4 = listenfd;
[2/httpd.c : 96:17] D.4750 = bind (listenfd.4, D.4748, D.4747);
[2/httpd.c : 96:12] if (D.4750 == 0)
  goto <bb 8>;
else
  goto <bb 9>;

<bb 8>:
[2/httpd.c : 96:9] option = {CLOBBER};
goto <bb 12>;

<bb 9>:
option = {CLOBBER};

<bb 10>:
[2/httpd.c : 90:29] p = [2/httpd.c : 90] p->ai_next;
```

شکل ۵ـ۲: نمونه‌ای از خروجی گراف جریان کنترلی نرم‌افزار GCC برای کد یک سرور http

در فایل متنی فوق bb به اختصار همان بلوک‌های ابتدایی می‌باشند. کد مربوط به هر بلوک ابتدایی، با ذکر نام فایل و شماره خط نظیر آن، زیر هر بلوک نشان داده شده است.

با پردازش فایل متنی بدست آمده، به طور تصادفی یک بلوک ابتدایی به عنوان محل فعال‌سازی و فراخوانی کد آسیب‌پذیر تزریق شده انتخاب می‌شود. لازم به ذکر است که امکان تزریق آسیب‌پذیری به یک عمق خاص و یا بلوک با بیشترین عمق و کمترین عمق نیز در برنامه تعبیه شده است.

۵ـ۲ ساخت توابع آسیب‌پذیر از روی بدنه‌ها

با توجه به اینکه تکه کدهای استخراج شده، از جنس بدنه‌ی بدون عمق می‌باشند، برای استفاده از آن‌ها، باید ابتدا آن‌ها را با افزودن هدر[۵] به تابع تبدیل کرد.

به منظور ایجاد گزینه‌ی پیچیده‌سازی بیشتر در تولید پیکره‌های آسیب‌پذیر، این امکان در نظر گرفته شده است تا بدنه‌ی ورودی به یک یا چند تابع تو در تو تبدیل شود. این امر می‌تواند باعث پیچیدگی بیشتر در کشف آسیب‌پذیری گردد.

```
1   int count, num[20];
2   count = 12;
3   int kp = num[count];
4   while (count < 20)
5       {
6           int dir_exec_id = count;
7           if (kp)
8           {
9               printf("Err: File not found");
10              int kp = -1;
11              count++;
12          }
13          memcpy(num[count], num[kp], sizeof(int));
14      }
15  num[2] = kp + count;
```

شکل ۵ـ۳: بدنه‌ی بدون عمق یک آسیب‌پذیری

با داشتن تکه کد فوق و افزودن هدرهای لازم، همچنین با بهره‌وری از امکان ذکر شده در فوق، کد بدون عمق فوق به دو تابع زیر تبدیل می‌گردد.

[۵] Header

```
1  void func_156315(void *count, void *kp, void *num){
2      int dir_exec_id = count;
3      if (kp)
4      {
5          printf("Err: File not found");
6          int kp = -1;
7          count++;
8      }
9      memcpy(num[count], num[kp], sizeof(int));
10 }
11
12 void func_176221(){
13     int count, num[20];
14     count = 12;
15     int kp = num[count];
16     while (count < 20)
17         {
18             func_156315(count, kp, num);
19         }
20     num[2] = kp + count;
```

شکل ۵ـ۴: تابع تولید شده با عمق ۲

همانطور که در شکل مشهود است، نه تنها بدنه‌ی بدون عمق به تابع تبدیل شده، بلکه قسمت داخل حلقه‌ی `while` نیز خود به عنوان یک تابع مجزا ایجاد شده است.

لازم به ذکر است که برای پیدا کردن نام متغیرهای محلی موجود در تکه کد، جهت ساخت تابع و پاس دادن صحیح آن‌ها از ابزار دیگری به نام `Ctags` استفاده شده است. این ابزار که یک پارسر ساده برای برنامه‌های سی و سی‌پلاس‌پلاس می‌باشد، می‌تواند نام توابع و متغیرها را با یک بار اجرا بر روی کد منبع، در قالب یک قایل متنی به کاربر تحویل دهد.

```
count           variable     1 vuln.c          int count, nums[20];
kp              variable     3 vuln.c          int kp = nums[count];
nums            variable     1 vuln.c          int count, nums[20];
```

شکل ۵ـ۵: خروجی نرم‌افزار Ctags بر روی بدنه‌ی آسیب‌پذیری.

۵ـ۳ فعال‌سازی آسیب‌پذیری

حال با داشتن یک تابع آسیب‌پذیر و همچنین یک نقطه‌ی حمله(بدست آمده به وسیله‌ی تحلیل گراف جریان کنترلی)، کافیست تابع مذکور را در خط مورد نظر فراخوانی کنیم. اگر مسیر اجرای برنامه، تمام شرط‌ها و پرش‌های شرطی لازم برای رسیدن به آن خط را با موفقیت طی کند، تابع آسیب‌پذیر فراخوانی شده و آسیب‌پذیر فعال می‌شود.

```
48  while (i < n1 && j < n2)
49  {
50      if (L[i] <= R[j]) {
51          arr[k] = L[i];
52          i++;
53      }
54      func_176221();
55      else {
56          arr[k] = R[j];
57          j++;
58      }
59      k++;
60  }
```

شکل ۵ـ۶: فراخوانی تابع آسیب‌پذیر در داخل یک حلقه‌ی while

یکی از اصلی‌ترین چالش‌ها در این بخش، وجود حلقه‌ها یا عبارات شرطی تک‌گزاره‌ای می‌باشد. می‌دانیم اگر مثلا بعد از یک `if` یا `while` فقط یک گزاره باشد، نیازی به گذاشتن آکولاد در اطراف آن گزاره نیست و آن گزاره مادامی که تنها یک خط باشد، خود به خود داخل حلقه یا عبارت شرطی به حساب می‌آید. این موضوع می‌تواند در هنگام فعال‌سازی آسیب‌پذیری مشکل‌ساز باشد. زیرا افزودن خط فراخوانی تابع آسیب‌پذیری باعث می‌شود گزاره‌ی قبلی به دلیل عدم وجود آلکولاد از حلقه یا عبارت شرطی بیرون بماند.

برای مقابله با این چالش و پیچیدگی‌های حاصل از آن، کد برای تشخیص چنین مواردی مورد بررسی قرار می‌گیرد و تنها خطوطی که افزودن فراخوانی تابع آسیب‌پذیر، خطری برای ماهیت برنامه ایجاد نمی‌کنند، به عنوان نقاط نامزد برای محل فعال‌سازی حمله انتخاب می‌شوند.

۵ـ۴ جزئیات بیشتر

ابزار پیاده‌سازی شده به زبان پایتون توسعه داده شده است و جهت تزریق و فعال‌سازی آسیب‌پذیری بر روی یک کد به زبان سی یا سی‌پلاس‌پلاس، با تمام نسخه‌های زبان سی و سی‌پلاس‌پلاس سازگار می‌باشد. این ابزار از دو ماژول[۶] اصلی و یک ماژول بالایی[۷] جهت مدیریت دو ماژول ذکر شده تشکیل شده است. در این بخش به تحلیل رفتاری هر یک از این ماژول‌ها می‌پردازیم.

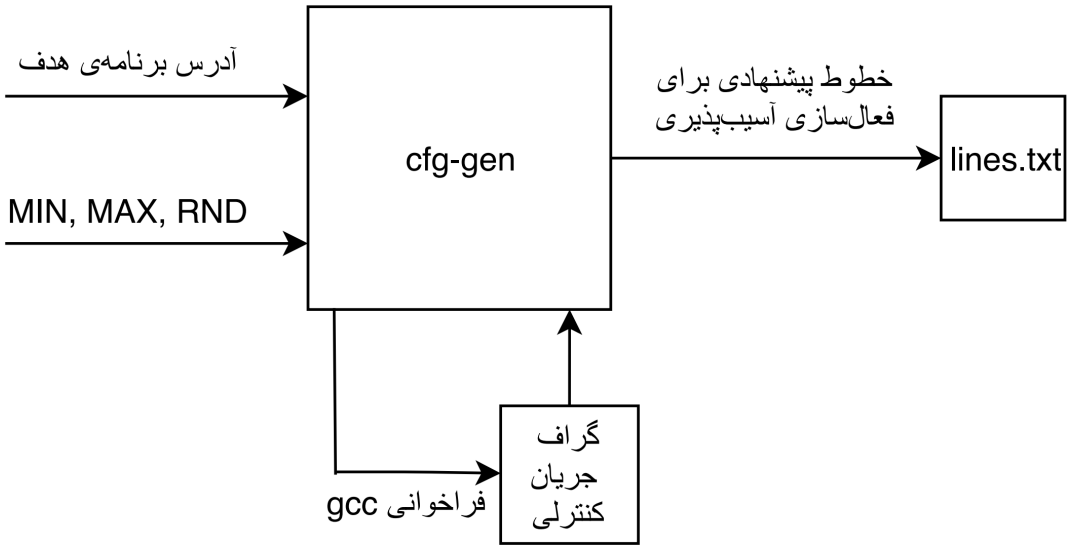
- تولید کننده‌ی گراف جریان کنترلی

این ماژول که در کد با نام `cfg-gen` استفاده می‌شود، مسئولیت ساخت و تحلیل گراف جریان کنترلی را که در بخش قبل به آن اشاره شد، دارد. شیوه‌ی عملکر آن به این صورت است که با فراخوانی ابزار gcc که جزئیات شیوه‌ی عملکرد آن در قسمت قبل ذکر شده است، اقدام به تولید گراف جریان کنترلی می‌کند. از آنجایی که خروجی gcc در قالب یک فایل در پوشه‌ی حاوی برنامه‌ی هدف ذخیره می‌شود و مستقیماً در دسترس این ماژول نمی‌باشد، `cfg-gen` ابتدا با جستجو در پوشه‌ی کد هدف، اقدام به یافتن فایل خروجی gcc می‌کند.

در مرحله‌ی بعد، این ماژول با بررسی فایل بدست آمده و همچنین با در نظر گرفتن گزینه‌ی عمق فعال‌سازی که می‌تواند `MAX`(فعال‌سازی در عمیق‌ترین بلاک ابتدایی برنامه)، `MIN`(فعال‌سازی در سطحی‌ترین بلوک ابتدایی برنامه) و یا `RND`(فعال‌سازی در بلوک ابتدایی با عمق تصادفی) باشد، اقدام به تحلیل بلوک‌های ابتدایی کد و تناظر آن‌ها با کد منبع کرده، سپس شماره‌ی چند خط از کد منبع را به عنوان نقاط ابتدایی نامزد برای فعال‌سازی آسیب‌پذیری، در یک فایل متنی با نام `line.txt` ذخیره می‌کند. چگونگی عملکرد این ماژول از منظر ورودی و خروجی در شکل زیر به صورت شماتیک تصویر شده است.

---

[۶] Module
[۷] Top



شکل ۵ـ۷: عملکرد ماژول cfg-gen بر اساس ورودی و خروجی‌ها

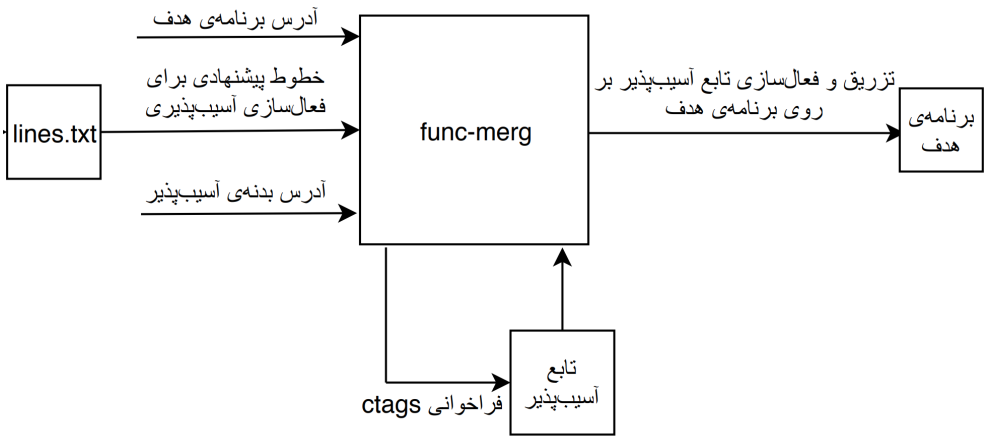
- ماژول تزریق کننده و فعال‌ساز آسیب‌پذیری

این ماژول که در کد با نام func-merg استفاده شده است، با دریاقت خطوط پیشنهادی از ماژول قبلی و حذف خطوط پر خطر از آن، همچنین با دریافت بدنه‌ی آسیب‌پذیر و تبدیل آن به یک یا چند تابع، اقدام به تزریق و سپس فعال‌سازی آسیب‌پذیری در برنامه‌ی هدف می‌کند.

برای تولید هر تابع از بدنه‌ی آسیب‌پذیر، این ماژول یک نام به صورت تصادفی برای تابع انتخاب کرده و هدرهای لازم را برای بدنه‌ی مذکور ایجاد می‌کند. همچنین همانطور که در بخش‌های قبلی به آن اشاره شد، برای تبدیل یک تابع به چند تابع، نیاز است متغیرهای تابع اولیه شناسایی شده و سپس به تابع بعدی پاس داده شوند. این امر در داخل این ماژول با بهره‌گیری از ابزار ctags از انجام می‌شود. خروجی ctags تحلیل شده و نام متغیرها از آن استخراج می‌شود.

تزریق آسیب‌پذیری با نوشتن توابع بدست آمده در انتهای فایل کد برنامه‌ی هدف و همچنین افزودن هدرهای آن‌ها در ابتدای همان فایل صورت می‌گیرد. این در حالیست که فعال‌سازی آسیب‌پذیری با فراخوانی توابع تزریق شده در یک خط منتخب درون جریان کنترلی و داده‌ای برنامه انجام می‌پذیرد.

چگونگی عملکرد این ماژول از منظر ورودی و خروجی در شکل زیر به تصویر شده است.



شکل ۵ـ۸: عملکرد ماژول `func-merg` بر اساس ورودی و خروجی‌ها

- ماژول بالایی

این ماژول وظیفه‌ی فراخوانی و مدیریت دو ماژول فوق را بر عهده دارد. همچنین کنترل حالت‌های استثنائی[۸] و گزارش پیغام‌های مناسب جهت عیب‌یابی نیز در حیطه‌ی وظایف این ماژول می‌باشد. علاوه بر این این ماژول نقش یک واسط بدون پیچیدگی را برای کاربر ایفا می‌کند. به گونه‌ای که تنها با دریافت آدرس برنامه‌ی هدف، آدرس بدنه‌ی آسیب‌پذیر و عمق مورد نظر برای فعال‌سازی، ماژول‌های دیگر و پیچیدگی روابط بین آن‌ها را مدیریت کرده و پس از اتمام فرایند فوق، فایل‌های اضافی که در مراحل مختلف ایجاد شده‌اند را پاک می‌کند.

[۸] Exception

فصل ۶

نتیجه‌گیری

از آن جایی که ارزشیابی ابزار ارائه شده، خود با پیچیدگی‌های زیادی همراه می‌باشد، به اجرا و ارزیابی این ابزار بر روی تعداد محدودی برنامه‌ی هدف بسنده شده است.

با توجه به نکات منظور شده در طراحی این ابزار، همچنین با بررسی نتایج محدود به دست آمده در چند اجرای آزمایشی، می‌توان انتظار داشت موارد ذکر شده در زیر در صورت ارزیابی این ابزار به صورت کامل، قابل مشاهده باشند.

- انتظار می‌رود که آسیب‌پذیری‌های تزریق شده کد جدید را غیر قابل اجرا نکرده و تغییری در ماهیت برنامه ایجاد نمی‌کنند. صرفاً یک رفتار آسیب‌پذیر به مجموعه رفتارهای برنامه افزوده می‌شود.
- آسیب‌پذیری‌های تزریق شده تنها در شرایطی خاص فعال شده و بدیهی نمی‌باشند. از این رو کشف آن‌ها با توجه به عمق فعال‌سازی آن‌ها با سختی‌های مختلفی همراه است.
- در بررسی‌های آتی با تعداد زیادی نمونه‌ی آزمایشی، انتظار می‌رود که فرایند تزریق و فعال‌سازی آسیب‌پذیری به تندی صورت پذیرفته و امکان اجرای آن بر روی تعداد بسیار زیادی کد منبع وجود داشته باشد.
- به دلیل عدم بهره‌وری از دانش خبره در تولید آسیب‌پذیری‌ها، انتظار می‌رود که پس از تحلیل عمیق کدهای آلوده شده، آسیب‌پذیری‌های تزریق شده دارای عمق و پیچیدگی تصادفی بوده و تمایل به نوع و میزان سختی خاصی نداشته باشند. از این رو معیاری عادلانه برای سنجش تمام ابزارهای

کشف آسیب‌پذیری به حساب خواهند آمد.

- آسیب‌پذیری‌های ایجاد شده کاملاً طبیعی بوده و بازتولید نشده‌اند. از این رو هیچ دانش خبره‌ای در تولید آن‌ها استفاده نشده است.

موارد فوق این ابزار را به یک ابزار قدرتمند، بدون نقاط ضعف ابزارهای قبلی و با نقاط قوت جدید تبدیل ساخته است، که می‌تواند برای تولید پیکره‌های معیار در سطح کلان جهت سنجش کیفیت عملکرد ابزارهای کشف آسیب‌پذیری، مورد استفاده قرار گیرد.

مراجع

[1] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. Lava: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121. IEEE, 2016.

[2] J. Pewny and T. Holz. Evilcoder: automated bug insertion. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 214–225. ACM, 2016.

[3] S. Roy, A. Pandey, B. Dolan-Gavitt, and Y. Hu. Bug synthesis: Challenging bug-finding tools with deep faults. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 224–234. ACM, 2018.

[4] P. Hulin, A. Davis, R. Sridhar, A. Fasano, C. Gallagher, A. Sedlacek, T. Leek, and B. Dolan-Gavitt. Autoctf: Creating diverse pwnables via automated bug injection. In *11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17)*, 2017.

Abstract

In spite of decades of research in vulnerability detection tools, there is a surprising dearth of ground-truth corpora that can be used to evaluate the efficacy of such tools. Therefore injecting vulnerability into source code with proper tagging and diversity in difficulty and kind, could make a great contribution to evaluation of vulnerability detection tools by creating diverse and novel ground-truth corpora. In this work, we present a novel approach towards injecting vulnerabilities into source code level by injecting vulnerabilities extracted from existing CVEs into arbitrary programs as a vulnerable function. We discuss that vulnerabilities injected by our tool are highly realistic under a variety of metrics and they do not favor a particular type.

Keywords: vulnerability, injection, source code, vulnerability detection

Sharif University of Technology

Department of Computer Engineering

B.Sc. Thesis

Vulnerability Injection on Source Code Level

By:

Seyed Parsa Tayefeh Morsal

Supervisor:

Dr. Kharrazi

August 2019