Driller: Augmenting Fuzzing Through Selective Symbolic Execution

ابزارهای جدید تنها می‌توانند به نقص‌های سطحی دست پیدا کنند. در مورد fuzzerها باید مکانیزمی برای تولید ورودی‌های جدید داشت که این ابزارها به صورت دلخواه این کار را انجام می‌دهند. در مورد ابزارهای با مکانیزم concolic ورودی‌های مختلف به کمک solver تولید می‌شود ولی مشکلی که دارند این است که با انفجار مسیر روبه‌رو می‌شوند. به همین دلیل نمی توانند به آسیب‌پذیری‌های در عمق بالا دست پیدا کنند.

دستاورد علمی این ابزار این است که روش fuzzing را با روش selective concolic ترکیب کرده است.

دو نوع ورودی تعریف می‌شود:

ورودی کلی:[۱] ورودی هست که می‌تواند بازه گسترده ای را در بر گیرد.

ورودی مشخص:[۲] یعنی باید از یک مجموعه محدود و مشخص یکی را انتخاب کرد.

وجود checkها روی همین ورودی‌ها باعث می‌شود که برنامه به تعدادی compartment تقسیم شود. جریان اجرایی بین این compartmentها خواهد بود.

مثلا برنامه ای را در نظر بگیرید که تعدادی command به عنوان ورودی دریافت می‌کند. ابتدا برنامه نام دستور را می‌خواند و با مجموعه نام‌های موجود بررسی می‌کند. سپس پارامترهای دیگر را بررسی می‌کند. برای Fuzzer تولید نام دستور مشکل است چون باید عبارتی مشخص مثل (ping) را تولید کند. ولی در ادامه در تولید پارامترهای ورودی آن دستور بهتر می‌تواند عمل کند(مثلا آدرس‌های ip مختلف را دلخواه تولید کند) در اینجا نام دستور، یک ورودی کلی و پرامترهای آن، یک ورودی مشخص هست.

بررسی Fuzzerها:

3 type of fuzzers:

mutating fuzzers:

With mutation, samples of valid input are mutated randomly to produce malformed input.

You can build in greater intelligence by allowing the fuzzer to do some level of parsing of the samples to ensure that it only modifies specific parts or that it does not break the overall structure of the input such that it is immediately rejected by the program. Some protocols or file formats will incorporate checksums that will fail if they are modified arbitrarily.

---

[۱] General Input

[۲] Specific Input

Two useful techniques that can be used by mutation-based fuzzers are described below:

Replay

A fuzzer can take saved sample inputs and simply replay them after mutating them. This works well for file format fuzzing where a number of sample files can be saved and fuzzed to provide to the target program.

Man-in-the-Middle or Proxy

MITM describes the situation where you place yourself in the middle of a client and server, intercepting and possibly modifying messages passed between them. In this way, you are acting like a proxy between the two.

Generation

Generation-based fuzzers actually generate input from scratch rather than mutating existing input. Generation-based fuzzers usually require some level of intelligence in order to construct input that makes at least some sense to the program, although generating completely random data would also technically be generation.

Generation fuzzers often split a protocol or file format into chunks which they can build up in a valid order, and randomly fuzz some of those chunks independently.

Evolutionary

Evolutionary fuzzing is an advanced technique. It allows the fuzzer to use feedback from each test case to learn over time the format of the input.

Compile-time instrumented fuzzing goes another route: It adds instructions to an application's code that allow the fuzzer to detect code paths in the application. It can then use promising fuzzing samples that expose large parts of the code for further fuzzing.

Anatomy of a fuzzer

To operate effectively, a fuzzer needs to perform a number of important tasks:

- Generate test cases
- Record test cases or any information needed to reproduce them
- Interface with the target program to provide test cases as input
- Detect crashes.

https://www.mwrinfosecurity.com/our-thinking/15-minute-guide-to-fuzzing/

ابزار Driller از ۴ قسمت اصلی تشکیل شده است:

۱. موردآزمون به عنوان ورودی: ابزار به صورت خودکار توانایی تولید موردآزمون را دارد ولی ورودی آن توسط کاربر می‌تواند به ابزار سرعت بخشد.

۲. Fuzzing: ابزار ابتدا با Fuzzing شروع به کار می‌کند. اگر به ورودی‌های «مشخص» برسد fuzzer گیر می‌کند.

۳. Concolic execution: وقتی fuzzer گیر کرد concolic execution شروع به کار می‌کند تا مسیر جدیدی را پیدا کند.

۴. Repeat: وقتی مسیر جدید پیدا شد، اجرا دوباره بهfuzzer سپرده می‌شود و اجرا ادامه پیدا می‌کند.

ویژگی‌های fuzzer:

۱. تولید ورودی‌ها بر اساس الگوریتم ژنتیک صورت می‌گیرد. تابع fitness در اینجا، میزان پوشش مسیرهای برنامه هست.

۲. دنبال کردن تغییر حالت برنامه: تمام جریان‌های کنترلی برنامه به شکل (منبع، مقصد) دنبال می‌شود. از این اطلاعات برای تابع fitness و تولید جمعیت جدید (ورودی جدید) استفاده می‌شود.

۳. اگر در مسیر اجرای برنامه حلقه وجود داشته باشد، یک محاسبه ثانویه انجام می‌شود که مشخص می‌کند که آیا این مسیر مناسب تولید جمعیت جدید هست یا نه! در این جا تعداد تکرارهای هر بار اجرای حلقه نگهداری می‌شود و اجراها در تعدادی سطل نگهداری می‌شود. هر اجرا در سطل مربوط به لگاریتم تعداد تکرارهایش قرار می‌گیرد. برای تولید جمعیت جدید این بار از هر سطل یک اجرا انتخاب می‌شود یعنی به جای N اجرا از log N اجرا استفاده می‌شود.

برای بر طرف کردن نقص fuzzerها از اجرای concolic استفاده می‌شود.

هرگاه fuzzer تعداد معینی از جهش‌ها را رفت و به حالت جدیدی نرسید گفته می‌شود که fuzzer «گیر» کرده است. سپس Driller ورودی که «جالب» تشخیص داده است را به اجرای concolic می‌دهد.

ورودی جالب است که:

۱: ورودی که باعث شود حالت‌های جدیدی از برنامه در مسیر اجرا کشف شود.

۲: ورودی که مسیر اجرایی آن اولین عضو یک «سطل از حلقه» هست.

تعریف ورودی «جالب»، تعداد مواردی که اجرا به موتور concolic داده می‌شود را کاهش می‌کند.

هرگاه fuzzer «گیر» کرد اجرای برنامه به موتور concolic داده می‌شود. دلیل گیر کردن این است که fuzzer نمی تواند قیدهای پیچیده را حل کند. تمام ورودی‌های «جالب» به موتور concolic داده می‌شود. با این ورودی‌ها برنامه اجرا می‌شود تا حالت‌هایی پیدا شوند که fuzzer نتوانسته اجرا کند. بعد از این که موتور concolic پردازش خود را تمام کرد، ورودی‌های جدیدی ایجاد می‌شود که به عنوان ورودی به fuzzer داده می‌شود و دوباره اجرا به fuzzer بازمی گردد.

اجرای concolic در Driller:

یک ویژگی مهم Driller است که وقتی اجرا به موتور concolic داده می‌شود، اجرای concolic دچار انفجار مسیر نخواهد شد. چون که fuzzer مسیر اجرای پیشین خود را به موتور concolic می‌دهد و اجرای concolic تنها سعی می‌کند با not کردن یکی از شرطهای مسیر ورودی از fuzzer به مسیری جدیدی برسد.

در اینجا Driller روی ورودی که از fuzzer می‌آید در موتور concolic یک قید تعریف می‌کند. تا همان مسیری که در آخرین اجرای fuzzer طی شده در اجرای concolic هم طی شود. بعد از این که در اجرای concolic به بلاک جدیدی از برنامه رسیدیم، قید مربوط به ورودی حذف می‌شود.

وجود تابع تولید مقدار دلخواه در کد برنامه می‌تواند روند fuzzer را مختل کند. بعد از یافتن آسیب‌پذیری در برنامه از اجرای نمادین استفاده می‌شود و حالت برنامه، هنگام توقف برنامه بررسی می‌شود تا رابطه بین ورودی و خروجی برنامه بدست آید. با این کار پروتکل challenge-response برنامه کشف می‌شود. بعد از پیدا کردن این رابطه می‌توان ورودی که موجب اکسپلویت برنامه می‌شود را تولید کرد.