Динамический анализ кода (также известный как «профилирование») наиболее популярен в областях, где требуется выявить hotspots приложения (пиковые с точки зрения производительности участки программы), гонки потоков (race conditions), найти ошибки при работе с памятью в приложении, оценить фактический расход оперативной памяти, а техники Taint Analysis и In-Memory Fuzzing (техники динамического анализа) позволяют определить какие участки программы наиболее подвержены exploit-ам.

В отличие от более распространенного (и простого в исполнении) статического анализа, динамический анализ обладает рядом преимуществ:

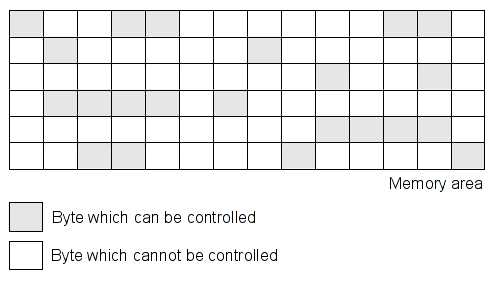
* Для проведения анализа исходный код программы не обязателен
* Наличие runtime информации (содержимое регистров, содержимое ячеек памяти, известны значения переменных окружения)
* Диагностика ошибок в многопоточном коде (таких, как борьба потоков за доступ к общим ресурсам, наличие deadlock-ов)
* Измерение потребляемых программой ресурсов (время исполнения программы или её отдельных частей, число обращений к внешним ресурсам, например, базе данных или файлу)
* Большая точность в диагностике false-positive ошибок: динамический анализатор не пытается предсказывать поведение программы, как это делает статический анализатор, а при запуске программы просто фиксирует наличие ошибки.

К очевидным недостаткам динамического анализа можно отнести:

* Невозможность покрыть 100% кода программы (покрывает только ту часть кода, которая выполняется)
* При наличии исходного кода программы и необходимости внести исправления, найти точное место в коде будет проблематично

TAINT ANALYSIS

Taint Analysis - популярный метод, который заключается в проверке того, какие переменные могут быть изменены пользователем. Любой пользовательский ввод может быть опасен, если он не проверен должным образом. С помощью этого метода можно проверить регистры и области памяти, которыми может управлять пользователь при сбое - это может быть полезно.



Например,

void foo1 ( const char \* av [])

{

uint32\_t a, b;

a = atoi ( av [ 1 ]);

b = а ;

foo2 ( b );

}

void foo2 ( const char \* av [])

{

uint8\_t \* buffer ;

if ( ! ( buffer = ( uint8\_t \* ) malloc ( 32 \* sizeof ( uint8\_t ))))

return ( - ENOMEM );

buffer [ 2 ] = av [ 1 ] [ 4 ];

buffer [ 12 ] = av [ 1 ] [ 8 ];

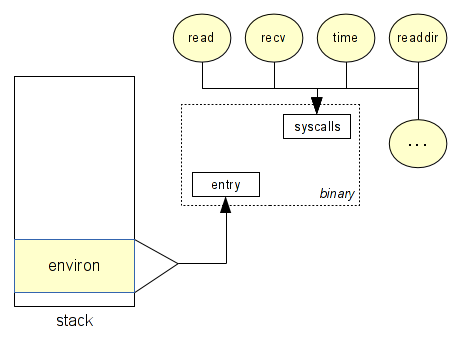
buffer [ 30 ] = av [ 1 ] [ 12 ];

}

В первой функции в начале переменные «a» и «b» не испорчены. Когда вызывается функция atoi, переменная «a» испорчена. Тогда «b» испорчена, если ей присвоено значение «a». Теперь мы знаем, что аргумент функции foo2 может управляться пользователем.

Во второй функции, когда буфер выделяется через malloc, содержимое не испорчено. Затем, когда выделенная область инициализируется пользовательским вводом, нам нужно испортить байты «буфер + 2», «буфер + 12» и «буфер + 30». Позже, когда один из этих байтов читается, мы знаем, что он может контролироваться пользователем.

Для динамического анализа в основном нам нужно определить все вводимые пользователем данные, такие как окружение и системные вызовы. Мы начинаем заражать эти входные данные и распространять/удалять это заражение, когда у нас есть такие инструкции, как GET/PUT, LOAD/STORE.



Для этого нам нужна структура динамического бинарного инструментария. Назначение DBI - добавить обработчик pre / post для каждой инструкции. Когда вызывается обработчик, мы можем получить всю необходимую информацию об инструкции или среде (памяти).

Подготовка

Для грамотного использования бинарной динамической инструментации, а главное использования в нужных местах, необходимо знать устройство программ (kernel и user space и тд), исполняемых файлов (сегменты кода, данных, heap, stack и тд), взаимодействие программ с операционной системой и её ядром (системные вызовы и тд), базовые понятия о работе компиляторов и их компоновщиков, так же для метода reversing, потребуются базовые знания ассемблера. Обычно это изучают в университетах на таких предметах, как «Операционные системы», «Компиляторы», «Язык ассемблера» и прочих. Но если не повезло с университетом или преподавателями, в интернете есть множество бесплатного материала по данным темам, например:

* в общих чертах про инструментацию, как статическую, так и динамическую

https://xakep.ru/2013/09/11/61232/

* немного поглубже в DBI

https://habr.com/ru/company/dsec/blog/142575/

* документация по использованию фреймворка для DBI Intel Pin

https://software.intel.com/sites/landingpage/pintool/docs/98314/Pin/html/index.html

* презентация одной конференции с основными терминами

https://docs.huihoo.com/blackhat/usa-2011/BH\_US\_11\_Diskin\_Binary\_Instrumentation\_Slides.pdf

* «если есть проблема, её уже решил какой-нибудь индус с ютуба»

https://youtube.com/playlist?list=PLKwUZp9HwWoDXHo51cWvYz1GRlM4dI5F2

* так как мы инструментируем приложения под линукс, неплохо прочитать книгу обо всём в этой ОС (у меня в программе обучения была эта книга, поэтому её и рекомендую)

Операционная система UNIX, 2-е издание  
Автор: Робачевский А., Немнюгин С., Стесик О.  
Год: 2010

* так же любые видео на ютубе с запросами «как работает компилятор», «как устроена программа на языке С/С++»
* курс по разработке модулей ядра, чтобы научиться самому с ним взаимодействовать

https://stepik.org/course/2051/promo#toc