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:

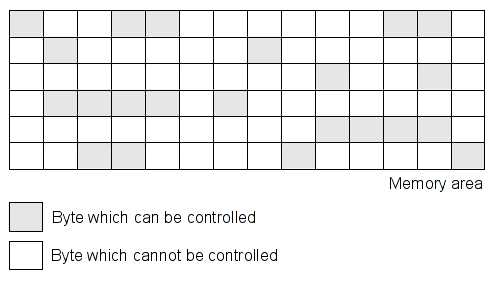
* Для проведения анализа исходный код программы не обязателен
* Наличие runtime информации (содержимое регистров, содержимое ячеек памяти, известны значения переменных окружения)
* Диагностика ошибок в многопоточном коде (таких, как борьба потоков за доступ к общим ресурсам, наличие deadlock-ов)
* Измерение 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 ];

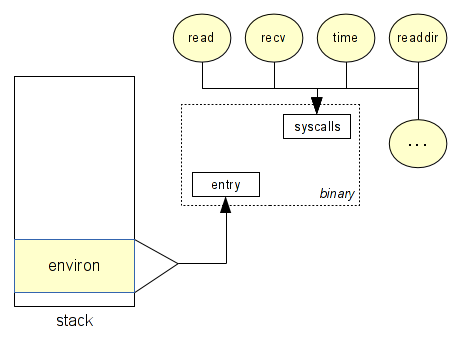
buffer [ 30 ] = av [ 1 ] [ 12 ];

}

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](https://stepik.org/course/2051/promo" \l "toc)

**Dynamic instrumentation tools**

Существует достаточно много инструментов для DBI, который позволяют в runtime манипулировать программой, её сегментами и используемыми регистрами. Суть таких фреймворков заключается в создании окружения выполнения программ и контролировании всех ресурсов, от уже упомянутых регистров и сегментов кода и данных, до процессов, потоков и каналов. В отличии от статического анализа мы проверяем только те инструкции, которые выполняются в зависимости от входных данных, поэтому DBI следует использовать в связке с фазингом, проверяя не только возможные уязвимости, но и возможные инструкции, появляющиеся в следствии ветвления.

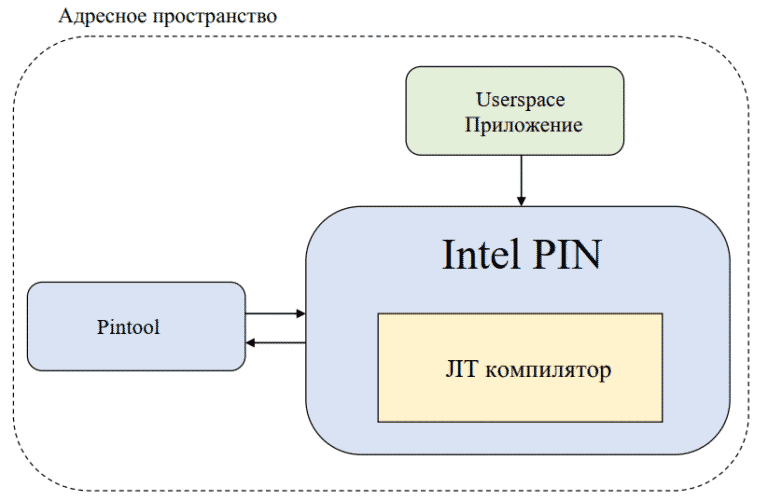
**Intel Pin**

Первое решение, найденное мне и подходящее под задачу (инструментирование приложений Linux без исходного кода) является Intel Pin.

Intel Pin — это DBI фреймворк для создания инструментов анализа userspace приложений. Возможности фреймворка позволяют управлять поведением уже скомпилированных программ, встраивая произвольный код, написанный на С/C++, прямо во время исполнения программы, то есть исходный код и перекомпиляция программы не требуются. Pin доступен для ОС Linux, Windows и OS X, также в официальных источниках можно встретить версию Pin для Android.

Инструменты анализа приложений, которые используют Pin API называются Pintools. Для разработки собственного инструмента Intel предоставляет набор разработчика, называемый Pin kits, куда также входят и исходные коды множества инструментов.

Pintool представляет из себя скомпилированный бинарный файл. Для Linux-систем — это разделяемая библиотека с расширением .so, для Windows – динамическая библиотека с расширением .dll. Взаимодействие Intel Pin, Pintool и анализируемой программы можно изобразить в виде схемы:



Intel Pin умеет работать в двух режимах — JIT и Probe. JIT (just-in-time) режим более функциональный, но менее производительный (наблюдается существенное замедление относительно исходной программы). Probe режим имеет более ограниченную функциональность, но производительность (связки ‘исходная программа’ + ‘Pintool’) приближается к производительности исходной программы.

Intel активно использует возможности фреймворка в своих продуктах, например: Intel® VTune ™ Amplifier, Intel® Inspector, Intel® Advisor и Intel® Software Development Emulator (Intel® SDE).

##### **Применение Intel PIN** **для различных сценариев диагностики уязвимостей ПО**

Intel Pin позволяет реализовать различные техники динамического анализа кода с целью диагностики потенциальных уязвимостей ПО. Приведу несколько сценариев из опыта моей команды:

* **Taint Analysis:** техника выявления потенциально уязвимых к пользовательскому вводу участков кода с целью обезопасить ПО от использования этих участков в интересах злоумышленников (защита ПО от так называемых exploit-ов). Базируется на идее присвоения каждому объекту в коде (переменной) специальной метки (тега), при условии, что значение этого объекта получено из ненадежного источника, т.е. от пользователя, из сети, из файла. Во время исполнения программы строится так называемое taint tree – метки сливаются, распространяются на другие объекты или удаляются в зависимости от инструкций GET/PUT, LOAD/STORE. Intel Pin позволяет добавить обработчики для каждой такой инструкции. На основе анализа taint tree возможно определить какие части программы потенциально подвержены exploit-ам, что позволяет лучше защитить такие участки от вредоносного воздействия. Если пользовательский ввод — это конфиденциальные сведения, то возможно сделать вывод о жизненном цикле такого типа данных внутри программы.
* **Dynamic** **Symbolic** **Execution** **(или** **Consolic** **Execution):** техника, применяемая для динамического покрытия кода для подтверждения того, что все ветви кода достижимы и работоспособны. Графически эту задачу можно представить как обход бинарного дерева, где узлы — это условные инструкции (conditional statements, т.е. if операторы), а ребра – последовательность не-условных инструкций (non-conditional statements), т.е. ветви кода. Очевидно, что при первом проходе любой из ветвей кода, значения входных параметров приведут либо к выполнению (true) или к невыполнению условия (false). Для следующего прохода вычисляются значения переменных, при которых условие будет противоположным. Задача Intel Pin добавить соответствующие обработчики для условных инструкций в коде. Фактически, применив данную технику вы покрываете весь код вашей программы с помощью test suite, который сгенерирован автоматически.
* **In-Memory Fuzzing:** техника тестирования потенциально уязвимых к пользовательскому вводу участков кода. Основная идея техники базируется на представлении, что почти любая программа будет сломана, если входные данные для неё будут подготовлены случайным образом. В начале выбирается участок кода программы, который необходимо протестировать. Затем определяется множество (диапазон) возможных значений входных параметров. Далее, в случае blackbox тестирования (когда ничего не известно об объекте тестирования) генерируется случайный набор входных параметров из возможного диапазона, для graybox (известна только часть) или whitebox (известно все) тестирования модифицируются известные входные наборы. Теперь участок кода исполняется и, если он приводит к непредсказуемому поведению, т.е. вызывает exception, то входной набор параметров запоминается и результаты анализируются. Далее шаги повторяются для нового набора входных данных. С помощью Intel Pin задача решается довольно просто: добавляются breakpoints до и после тестируемого участка кода; после прохождения первого breakpoint сохраняется контекст и отслеживаются все exceptions в коде. В случае, если произошел exception, набор входных параметров сохраняется и контекст восстанавливается, в случае если exception не произошел — просто восстанавливается контекст. Данная техника позволяет получить устойчивый к пользовательскому вводу код. Один из вариантов применения техники на практике – тестирование parser-а бинарного файла.

**Простой пример использования Pin**

В этом первом примере мы испортим область памяти "чтение" и увидим краткий обзор API Pin. Для этого первого испытания мы будем :

1. Ловить sys\_read syscall.

2. Получим второй и третий аргумент для taint области.

3. Вызовем хэндлер, когда у нас есть такая инструкция, как LOAD или STORE в этой области.

Мы будем работать над этим следующим кодом, функция foo выполняет простые LOAD и STORE инструкции.

void foo(char \*buf){

char a;

a = buf[0];

a = buf[4];

a = buf[8];

a = buf[10];

buf[5] = 't';

buf[10] = 'e';

buf[20] = 's';

buf[30] = 't';

}

int main(int ac, char \*\*av)

{

int fd;

char \*buf;

if (!(buf = malloc(256)))

return -1;

fd = open("./file.txt", O\_RDONLY);

read(fd, buf, 256), close(fd);

foo(buf);

}

С помощью Pin можно добавить обработчик pre и post при возникновении syscall. Для этого нам просто нужно инициализировать функцию обратного вызова.

typedef VOID(\* LEVEL\_PINCLIENT::SYSCALL\_ENTRY\_CALLBACK)(THREADID threadIndex, CONTEXT \*ctxt, SYSCALL\_STANDARD std, VOID \*v);

typedef VOID(\* LEVEL\_PINCLIENT::SYSCALL\_EXIT\_CALLBACK)(THREADID threadIndex, CONTEXT \*ctxt, SYSCALL\_STANDARD std, VOID \*v);

VOID LEVEL\_PINCLIENT::PIN\_AddSyscallEntryFunction(SYSCALL\_ENTRY\_CALLBACK fun, VOID \*val);

VOID LEVEL\_PINCLIENT::PIN\_AddSyscallExitFunction(SYSCALL\_EXIT\_CALLBACK fun, VOID \*val);

В нашем случае мы используем клиент LEVEL\_PINCLIENT::PIN\_AddSyscallEntryFunction. Когда происходит системный вызов, мы проверяем, является ли системный вызов доступным для чтения. Затем мы сохраняем второй и третий аргументы, которые описывают нашу область памяти.

struct range

{

UINT64 start;

UINT64 end;

};

std::list<struct range> bytesTainted;

VOID Syscall\_entry(THREADID thread\_id, CONTEXT \*ctx, SYSCALL\_STANDARD std, void \*v)

{

struct range taint;

if (PIN\_GetSyscallNumber(ctx, std) == \_\_NR\_read){

/\* Get the second argument \*/

taint.start = static\_cast<UINT64>((PIN\_GetSyscallArgument(ctx, std, 1)));

/\* Get the third argument \*/

taint.end = taint.start + static\_cast<UINT64>((PIN\_GetSyscallArgument(ctx, std, 2)));

/\* Add this area in our tainted bytes list \*/

bytesTainted.push\_back(taint);

/\* Just display information \*/

std::cout << "[TAINT]\t\t\tbytes tainted from " << std::hex << "0x" << taint.start \

<< " to 0x" << taint.end << " (via read)"<< std::endl;

}

}

int main(int argc, char \*argv[])

{

/\* Init Pin arguments \*/

if(PIN\_Init(argc, argv)){

return Usage();

}

/\* Add the syscall handler \*/

PIN\_AddSyscallEntryFunction(Syscall\_entry, 0);

/\* Start the program \*/

PIN\_StartProgram();

return 0;}

Теперь нам нужно поймать все инструкции, которые читают (ЗАГРУЖАЮТ) или записывают (ХРАНЯТ) в зараженной области. Для этого мы добавим функцию, вызываемую каждый раз, когда делается доступ к этой области.

Для этого мы добавим мастер-обработчик, вызываемый для каждой инструкции.

typedef VOID(\* LEVEL\_PINCLIENT::INS\_INSTRUMENT\_CALLBACK)(INS ins, VOID \*v); VOID LEVEL\_PINCLIENT::INS\_AddInstrumentFunction(INS\_INSTRUMENT\_CALLBACK fun, VOID \*val);

Затем в главном обработчике нам нужно найти инструкцию LOAD / STORE, например mov rax, [rbx] или mov [rbx], rax.

if (INS\_MemoryOperandIsRead(ins, 0) && INS\_OperandIsReg(ins, 0)){

INS\_InsertCall(

ins, IPOINT\_BEFORE, (AFUNPTR)ReadMem,

IARG\_ADDRINT, INS\_Address(ins),

IARG\_PTR, new string(INS\_Disassemble(ins)),

IARG\_MEMORYOP\_EA, 0,

IARG\_END);

}

else if (INS\_MemoryOperandIsWritten(ins, 0)){

INS\_InsertCall(

ins, IPOINT\_BEFORE, (AFUNPTR)WriteMem,

IARG\_ADDRINT, INS\_Address(ins),

IARG\_PTR, new string(INS\_Disassemble(ins)),

IARG\_MEMORYOP\_EA, 0,

IARG\_END);

}

int main(int argc, char \*argv[])

{

...

INS\_AddInstrumentFunction(Instruction, 0);

...

}

Как вы можете видеть выше, мы делаем некоторые проверки, прежде чем вставить вызов. Если инструкции второго операнда считываются в память и если первый операнд является регистром. То есть если инструкция выглядит как " mov reg, [r/imm]", она вызывает функцию ReadMem. При вызове этой функции она передает несколько сведений, таких как: адрес инструкции, дизассемблирование и адрес считываемой памяти. То же самое с инструкциями STORE.

Теперь нам просто нужно проверить, находится ли доступная память в зараженной области.. Для нашего обратного вызова чтения памяти у нас есть что-то вроде этого:

VOID ReadMem(UINT64 insAddr, std::string insDis, UINT64 memOp)

{

list<struct range>::iterator i;

UINT64 addr = memOp;

for(i = bytesTainted.begin(); i != bytesTainted.end(); ++i){

if (addr >= i->start && addr < i->end){

std::cout << std::hex << "[READ in " << addr << "]\t" << insAddr

<< ": " << insDis<< std::endl;

}

}

}

Конечный результат выглядит примерно так:

$ ../../../pin -t ./obj-intel64/Taint\_ex1.so -- ./test\_ex1

[TAINT] bytes tainted from 0x665010 to 0x665110 (via read)

[READ in 665010] 400620: movzx eax, byte ptr [rax]

[READ in 665014] 40062a: movzx eax, byte ptr [rax+0x4]

[READ in 665018] 400635: movzx eax, byte ptr [rax+0x8]

[READ in 66501a] 400640: movzx eax, byte ptr [rax+0xa]

[WRITE in 665015] 40064f: mov byte ptr [rax], 0x74

[WRITE in 66501a] 40065a: mov byte ptr [rax], 0x65

[WRITE in 665024] 400665: mov byte ptr [rax], 0x73

[WRITE in 66502e] 400670: mov byte ptr [rax], 0x74