межею такої стратегії буде повернення до послідовної системи.

Аналіз наведених факторів показує, що програмні витрати, витрати на диспетчеризацію можна віднести до послідовної частини алгоритму розв'язання задачі, витрати через дисбаланс завантаженості процесорів, комунікаційні витрати – до паралельної частини алгоритму.

Висновки

1. Розрахунки показують, що значення прискорень обчислень при використанні паралелізму за залежностями (1) та (2) мають істотні відмінності (рис. 1, 2), які збільшуються із збільшенням в системі кількості процесорів, що працюють паралельно.

2. Коментарі до закону Амдала про те, що обсяг задачі при зміні кількості процесорів залишається незмінним [2] – некоректні. Фактично залежність Амдала (2) враховує однакову пропорцію збільшення як послідовної, так і паралельної складової, що зумовлює (і саме про це засвідчує) стале значення параметра *f*.

3. Для визначення прискорення обчислень за рахунок застосування паралелізму перевагу потрібно надавати залежності (2) Амдала, тому що порівняно із законом Густафсона (3) ця залежність критичніша до процесів в ОС, пов'язаних із нарощуванням кількості процесорів.

1. Мельник А. О. Архітектура комп'ютера. Наукове видання. – Луцьк: Волинська обласна друкарня, 2008. – 470 с. 2. Цилькар Б. Я., Орлов С. А. Организация ЭВМ и систем: Учебник для вузов. – СПб.: Питер, 2004. – 668 с. 3. Тербер К. Дж. Архитектура высокопроизводительных вычислительных систем. – М.: Наука, 1985. – 272 с.

УДК 681.3

РЕЗУЛЬТАТИ ДОСЛІДЖЕНЬ ТА РЕАЛІЗАЦІЇ МЕХАНІЗМУ ПРИХОВАНОГО ЗБИРАННЯ ІНФОРМАЦІЇ З СИСТЕМ ВІРТУАЛЬНИХ ПРИМАНОК

© Назар Тимошик, Роман Тимошик, 2009

Національний університет "Львівська політехніка", кафедра захисту інформації,
вул. С. Бандери, 12, 79013, Львів, Україна

Наведено результати досліджень та реалізації механізму інтроспекційного аналізу системи-приманки, який забезпечує стійкий до виявлення та блокування моніторинг діяльності зловмисника в операційній системі Linux. Детально розглянуто труднощі реалізації та особливості архітектурної реалізації. На основі реалізованого програмного забезпечення уможливлюється повний та ефективний контроль подій у віртуалізованій ОС.

Приведено результаты исследований и реализации механизма интроспекционного анализа системы-приманки, который обеспечивает стойкий к выявленню и блокированию мониторинг деятельности злоумышленника в операционной системе Linux. Детально рассмотрены трудности реализации и особенности архитектурной реализации. На основании реализованного программного обеспечения стает возможным полный и эффективный контроль событий в виртуализированной ОС.

The results of researches and realization of mechanism of introspection analysis of the honeypot systems, which provides proof to the exposure and blocking monitoring of activity of malefactor in the operating system of Linux, are resulted in the article. Difficulties of realization, and features of architectural realization, are considered in detail. On the basis of the realized software complete and effective control of events becomes possible in vitrualized OS.

Особливості інтроспекційного аналізу. У роботах [1, 2] описувались важливість та актуальність новітніх стійких до блокування зловмисником механізмів збору інформації з систем-приманок, недоліки відомих нині рішень та був виконаний огляд можливих способів вирішення цієї проблеми. У цій статті пропонуються результати досліджень та реалізації механізму Virtual Machine Introspection (VMI) [3], завданням якого є

моніторинг внутрішнього стану операційної системи через монітор віртуальних машин (hypervisor/VMM). Основною перевагою цього механізму порівняно з іншими запропонованими рішеннями є прихованість та стійкість до блокування зловмисником, а також повнота даних, які можливо отримувати для різноманітних систем захисту інформації, наприклад, для антивірусних систем та систем контролю стану. Нами реалізовано базовий функціонал інтроспекційного аналізатора з можливістю перехоплення дій зловмисника на системах-приманках.

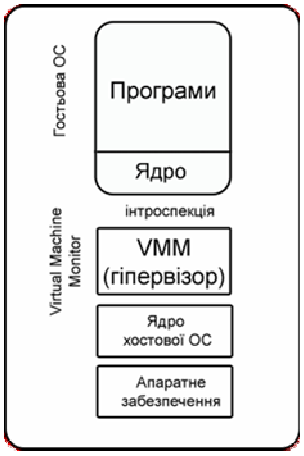

Рис. 1. Схема взаємодії між віртуалізованою ОС та гіпервізором

Для реалізації механізму ми використали відкритий гіпервізор Qemu [4], який найчастіше використовується у відкритих та закритих проектах віртуалізації завдяки своїй простоті, легкості та можливості трансляції коду. В основі гіпервізора Qemu лежить динамічний транслятор, який адаптує інструкції, що передаються з гостьової віртуальної операційної системи хостовій операційній системі. При цьому можливе використання програмного акселератора kqemu, який являє собою модуль ядра і транслює інструкції напряму від qemu до ядра хостової машини. Це дає змогу значно підвищити продуктивність системи на ядрі операційної системи (ОС) Linux 2.4, але при використанні ядра 2.6 приріст продуктивності незначний. Загалом схему перехоплення подій з віртуалізованої ОС подано на рис.1.

Перевагою запропонованого підходу перед відомими нині рішеннями [1] є відсутність необхідності вносити зміни у вихідний код ядра гостьової чи хостової ОС, неможливість виявити та заблокувати засоби моніторингу, можливість віртуалізації більшості відомих процесорних архітектур.

Системні виклики є прошарком між апаратним забезпеченням та процесами, що працюють у просторі користувача. Цей прошарок призначений для трьох головних цілей. По-перше, він забезпечує абстрактний інтерфейс між апаратурою та простором користувача. Наприклад, під час запису або читання даних з файла для прикладних програм не має значення тип жорсткого диска, середовище, носій інформації, і навіть файлова система, на якій розміщений файл. По-друге, системні виклики гарантують безпеку і стабільність системи. Оскільки ядро працює посередником між ресурсами системи та простором користувача, то воно може приймати рішення про надання доступу відповідно до прав користувачів та інших критеріїв. Наприклад, це дає змогу запобігти можливості неправильного використання апаратних ресурсів програмами, захоплення будь-яких ресурсів у інших програм, а також можливість заподіяння шкоди системі. І, нарешті, один загальний шар між простором користувача та рештою частин системи надає можливість здійснити віртуальне управління процесами. В операційній системі Linux системні виклики є єдиним засобом, завдяки якому користувацькі програми можуть взаємодіяти з ядром; вони є єдиною законною точкою входу в ядро. Інші інтерфейси ядра, такі, як файли пристроїв або файли на файловій системі /proc зводяться до звернення через системні виклики. Таким механізмом, який може подати сигнал ядру, є програмне переривання – створюється виняткова ситуація (exception) і система перемикається в режим ядра для виконання обробника цієї виняткової ситуації. Обробник виняткової ситуації у цьому випадку і є обробником системного виклику (system call handler). Для апаратної платформи x86 це програмне переривання визначено як інструкція процесора int $ 0x80 [5]. Вона приводить в дію механізм перемикання в режим ядра і виконання вектора виняткової ситуації з номером 128, який є обробником системних викликів system_call (), що залежить від апаратної платформи і визначений у файлі entry.S. Функція system_call() перевіряє правильність переданого номера системного виклику, порівнюючи його зі значенням сталої NR_syscalls. Якщо значення номера більше або дорівнює значенню NR_syscalls, то функція повертає значення - ENOSYS. В іншому випадку викликається відповідний системний виклик: call * sys_call_table (,%

eax, 4). Оскільки кожен елемент таблиці системних викликів має довжину 32 біти (4 байти), то ядро множить цей номер системного виклику на 4 для отримання потрібної позиції в таблиці системних викликів.

Для нових процесорів з'явилася нова інструкція sysenter [5]. Ця функція забезпечує більш швидкий та спеціалізований спосіб входу в ядро для виконання системного виклику, ніж використання інструкції програмного переривання - int. Підтримка такої функції була додана в ядро 2.6.x. Незалежно від того, як виконується системний виклик, основним є те, що простір користувача зумовлює виключну ситуацію, або переривання, щоб викликати перехід в ядро. Цікаво, що в ОС Linux реалізовано значно менше системних викликів, ніж у багатьох інших операційних системах.

Згідно з [3] процедура перехоплення подій (інтроспекційного аналізу) у віртуалізованому середовищі складається з таких кроків:

1) Виокремлення функцій, що беруть участь в обробці переривань та використовуються для відлагодження гіпервізора та роботи з пам'яттю віртуальної машини:

− перехоплення інструкцій, що передаються від віртуалізованої ОС до гіпервізора;

− аналіз інструкцій;

− збереження результатів аналізу.

2) Аналіз вхідної інформації:

− аналіз інструкцій з гіпервізора;

− аналіз аргументів системних викликів.

3) Аналіз структур ядра ОС Linux:

− з виокремленням адрес вказівників (структура mm_struct) сегментів даних, сегментів коду та сегментів стека в оперативній пам'яті віртуалізованої ОС;

− з виокремленням адрес вказівників на запущені наступний, попередній, назви процесу і його стан (структура task_struct).

В умовах поточної стабільної версії ядра Linux 2.4 є близько 230 системних викликів, деякі найважливіші з них, які найчастіше використовуються і цікаві для перехоплення засобами інтроспекційного аналізатора, наведено в табл.1.

Таблиця 1

Опис основних системних викликів ОС Linux

Назва системного виклику	Опис	ID
sys_read	Використовується для читання з файлів	3
sys_write	Використовується для запису у файли	4
sys_open	Використовується для створення чи відкриття файлів	5
sys_getdents/sys_getdents64	Використовується для отримання лістингу вмісту директорії (так, як і /proc)	141/220
sys_socketcall	Використовується для керування сокетами	102
sys_query_module	Використовується для запиту завантажених модулів ядра	167
sys_setuid/sys_getuid	Використовується для керування UIDs	23/24
sys_execve	Використовується для виконання бінарних файлів	11
sys_chdir	Використовується для зміни поточної директорії	12
sys_fork/sys_clone	Використовується для створення процесу-нащадка	2/120
sys_ioctl	Використовується для роботи з пристроями	54
sys_kill	Використовується для відправки сигналів процесам	37

Qemu виконує трансляцію інструкцій процесору блоками [4]. Завдяки добре продуманому транслятору оператори, що працюють з пам'яттю, зокрема зі стеком, заміняються на відповідні аналоги з перевіркою границь та корекцією адрес.

Принцип виконання системних викликів полягає в занесенні аргументів потрібного виклику в регістри та виконання переривання (інструкція 0xCD). Виконання системних викликів відбувається так:

1. У регістр EAX заноситься номер системного виклику.
2. У решту регістрів EBX, ECX, EDX, EDI, EBP та ESI заносяться значення аргументів відповідної функції.
3. Виконується виклик переривання (int 0x80).
4. Значення регістрів процесора зберігаються в стек.
5. Ядро ОС виконує виклик за адресою sys_call_table+EAX.
6. Значення регістрів процесори витягуються із стека.
7. Результат системного виклику заноситься в регістр EAX.
8. Виконується інструкція повернення управління процесом.

Наведемо фрагмент коду трансльованого гіпервізором з викликом sys_execv у режимі in_assm:

```
----------
IN:
0xc010761c:  sub    $0x8,%esp
0xc010761f:  mov    %ebx,(%esp,1)
0xc0107622:  mov    %esi,0x4(%esp,1)
0xc0107626:  mov    %edx,%ebx
0xc0107628:  mov    %eax,%esi
0xc010762a:  mov    %ecx,%edx
0xc010762c:  mov    $0xb,%eax
0xc0107631:  mov    %ebx,%ecx
0xc0107633:  push   %ebx
0xc0107634:  mov    %esi,%ebx
0xc0107636:  int    $0x80
```

При реалізації проекту використовувались не тільки рідні структури Qemu, але й структури, що містяться у ядрі Linux. До останніх належать task_struct, mm_struct та vm_area_struct. Знання цих структур необхідне для коректного порівняння системного виклику і процесу, що його виконав.

Основною структурою, яка використовувалась при реалізації проекту, була структура CPUState, що дає прямий доступ до регістрів віртуального процесора (базових, прапорців, FPU, MMX тощо), які мають вигляд декількох масивів:

1. regs – базові регістри.
2. segs – сегменти.
3. fptags – регістри стану FPU.
4. fpregs – регістри стану FPU.
5. xmm_regs – внутрішні змінні емулятора.
6. dr – регістри відлагодження.

Отже, можливо інтерпретувати такі фрагменти бінарного транслятора:

```
73: v=80 e=0000 i=1 cpl=0 IP=0010:c0107443 pc=c0107443 SP=0018:c036bfd0 EAX=00000078
EAX=00000078 EBX=00010f00 ECX=c0105030 EDX=00000000
ESI=c036bfd0 EDI=c0105000 EBP=0008e000 ESP=c036bfd0
EIP=c0107443 EFL=00000202 [-------] CPL=0 II=0 A20=1 SMM=0 HLT=0
ES =0018 00000000 ffffffff 00cf9300
CS =0010 00000000 ffffffff 00cf9a00
SS =0018 00000000 ffffffff 00cf9300
DS =0018 00000000 ffffffff 00cf9300
FS =0000 00000000 00000000 00000000
GS =0000 00000000 00000000 00000000
LDT=00a8 c0325e20 00000027 c0008232
TR =00a0 c03a2800 000000eb c000893a
GDT=     c0324f00 0000011f
IDT=     c03a2000 000007ff
CR0=80050033 CR2=00000000 CR3=00101000 CR4=00000690
CCS=0000000c CCD=00010f00 CCO=LOGICL
```

На 73 перериванні відбувається системний виклик (на це вказує вектор переривання 80) відбувається системний виклик sys_close (значення в регістрі EAX=120). Інтерпретацію цього фрагмента виконання системного виклику ми виконуємо з файла helper.c у функції do_interrupt:

```
....
void do_interrupt(int intno, int is_int, int error_code,
                  target_ulong next_eip, int is_hw)
{
if ((intno == 0x80)&&(is_int == 1)) log_syscall_main(env);
   if (loglevel & CPU_LOG_INT) {
      if ((env->cr[0] & CR0_PE_MASK)) {
         static int count;
         fprintf(logfile, "%6d: v=%02x e=%04x i=%d cpl=%d IP=%04x:" TARGET_FMT_lx " pc=" TARGET_FMT_lx "
SP=%04x:" TARGET_FMT_lx,
                  count, intno, error_code, is_int,
                  env->hflags & HF_CPL_MASK,
                  env->segs[R_CS].selector, EIP,
                  (int)env->segs[R_CS].base + EIP,
                  env->segs[R_SS].selector, ESP);
....
```

Для побудови дерева процесів використовувалась згадана вище структура task_struct [5]. У цій структурі можна виокремити чотири важливі змінні, а саме унікальний ідентифікатор процесу (PID), назву процесу (залежно від версії ядра 16 або 20 символів), вказівники на попередній (prev_task) та наступний процес (next_task). На перший батьківський процес ОС Linux посилається вказівник init_task_union (рис. 2), адреса якого зберігається в файлі System.map, що створюється при компіляції ядра ОС [4]. У цьому самому файлі міститься вказівник на початок таблиці системних викликів sys_call_table. Величина структури task_struct залежить від версії ядра та може коливатись від 2 Кб до 16 Кб. Максимальна кількість процесів, що можуть бути одночасно запущені на ОС Linux, обмежується залежно від версії ядра і сумарного об'єму пам'яті, що виділяється під процеси (для ядра 2.4 це – 3 Гб). Перший процес має назву swapper. Під процеси, що виконуються в ОС Linux, виділяються окремі сегменти пам'яті. Регістр EIP містить адресу останньої трансльованої команди із сегмента, вказаного в регістрі CS. Спеціальні внутрішні регістри гіпервізора містять адреси на початок і кінець блоків трансляції (sysenter, sysexit) самого гіпервізора. Знаючи адресу CS:EIP перед системним викликом і маючи вказівники на структуру mm_struct кожного процесу, можна порівняти окремі системні виклики та процеси, що їх ініціювали, через приналежність адреси CS:EIP до сегмента даних окремого процесу.

Ядро подає адресний простір процесу у вигляді структури даних, яка називається дескриптором пам'яті. Ця структура містить всю інформацію, яка відноситься до адресного простору процесу. Дескриптор пам'яті формується за допомогою структури struct mm_struct, яка визначена у файлі <linux/sched.h>. Для прямої роботи із пам'яттю використовують сегмент та зміщення. Сегмент являє собою адресу виділеного блока пам'яті, доступ до якого не заборонений (захищений режим). Виділяють сегмент даних, сегмент коду та сегмент стеку [4]. Вказівники на поточні сегменти зберігаються у відповідних регістрах процесора EIP, DS, CS та SS. У захищеному режимі ядро стежить за тим, щоб сегмент даних та сегмент стека були захищені від запуску. В ОС Linux сегменти пам'яті ядра та користувацькі сегменти пам'яті розділені.

При прямому доступі до пам'яті віртуальної машини необхідно враховувати зміщення початку віртуальної пам'яті відносно фізичної [6]. При копіюванні даних в деяких випадках є певні нюанси, які потрібно враховувати, як наприклад, довжина буфера при доступі до даних через вказівник на рядок (змінна типу char *). Якщо через один з аргументів не передається довжина рядка, розмір ділянки пам'яті, який потрібно скопіювати, невідомий.

Розроблена нами програма аналізує вхідні дані у вигляді значень переривань та адрес пам'яті, аналізує системні виклики та структури даних і зберігає необхідні результати у файл (рис. 3). У цьому прикладі коду ми демонструємо інтерпретацію системного виклику #11 sys_execve, який відповідає за виконання бінарних програм. Фактично завдяки їй ми зможемо відслідкувати кожну виконану зловмисником команду, а через виклик sys_write побачити результат її виконання.

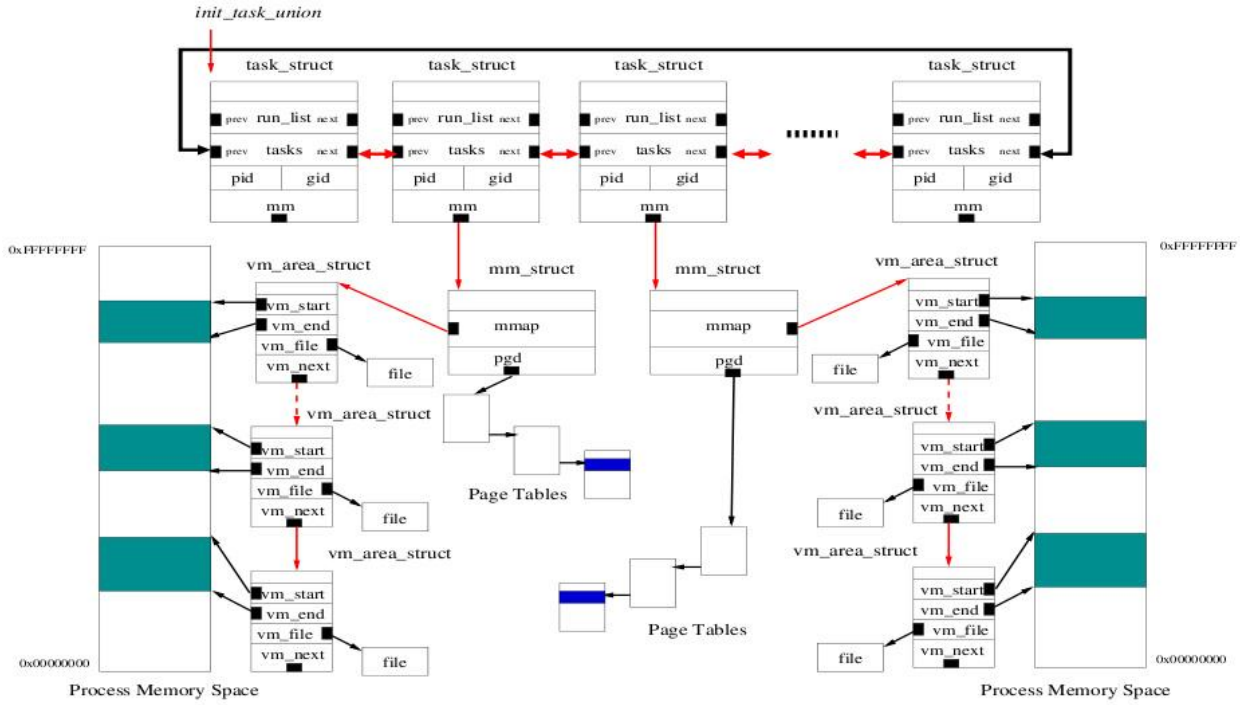

Рис.2. Подання процесів у пам'яті ОС Linux[5]

Внесенням необхідних змін у вихідний код гіпервізора Qemu було досягнуто поставлене завдання без істотного зниження продуктивності. З огляду на структурний підхід при виконанні гіпервізора було недоцільно комбінувати його з об'єктно-орієнтованим підходом. Тому всі модифікації виконано у вигляді окремих функцій. Наведемо приклад коду для отримання події виконання програми у віртуалізованій ОС через розроблену нами функцію log_syscall_main:

```
void log_syscall_main(CPUState *env)
{
....
switch(env->regs[R_EAX]) {
case 11:
        cpu_memory_rw_debug(env,env->regs[R_EBX],buf[0],64,0);
fprintf(syscall_list,"sys_execve(%s)\t\t\n",buf[0]);
....
}
```

Тестування системи. Практичне тестування програмного забезпечення виконувалось на операційній системі Novell SLES 10.1(ядро версії 2.6.16) та Novell OpenSuse 11(ядро 2.6.24). Як віртуалізована операційна система при тестуванні програмного забезпечення використовувалась ОС RedHat 8 (ядро 2.4.18). Усі тестування здійснювались на наступній конфігурації AMD Athlon 2600+, 1,25 Гб ОЗП. Для функціонування віртуальної ОС виділялось 256 Мб ОЗП.

В результаті тестувань було зафіксовано всі виконані зловмисником команди (через перехоплення виклику sys_execve), відкриті файли (sys_open/sys_close), породжені процеси та модифіковані ділянки пам'яті (див. рис. 3), а також результати виконання цих програм, які були виведені на консоль зловмисника (sys_write).

Для визначення продуктивності системи використовувались утиліти та встановлене програмне забезпечення, наведене у табл. 2. Зниження продуктивності гіпервізора після реалізації проекту становило не більше ніж 5–7 % порівняно з оригінальною версією (рис. 4).

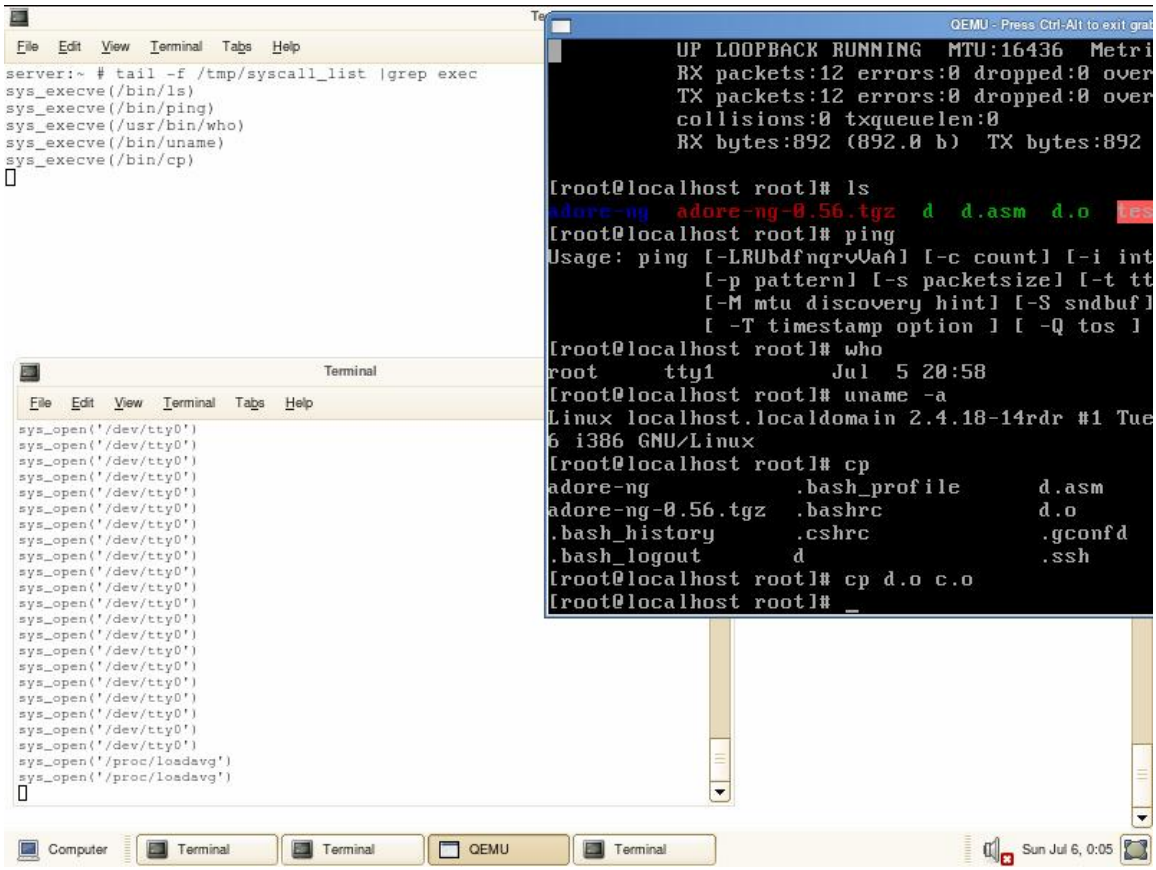

Рис.3. Моніторинг діяльності зловмисника на консолі системи-приманки

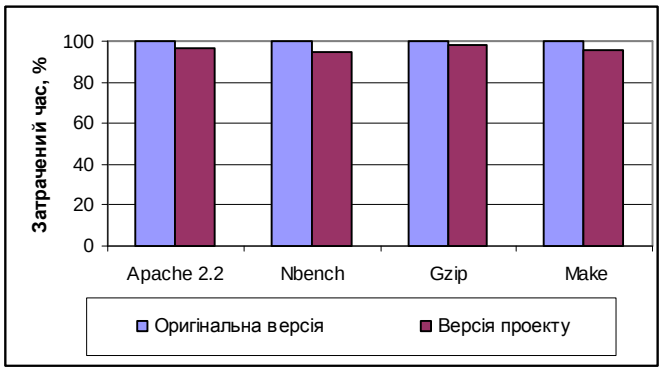

Рис. 4. Порівняльна діаграма продуктивності оригінального гіпервізора Qemu та версії проекту

Таблиця 2

Засоби вимірювань для розрахунку втрат продуктивності системи

Програмне забезпечення	Версія	Конфігурація
Веб-сервер Apache	2.2	Типова конфігурація із запуском у режимі Apache Worker MPM
Nbench	2.2.2	Типова конфігурація
GZip	1.3.3	Архівація файла (256 Мб)
Make	3.8.0	Компіляція ядра 2.6.15

Висновки. У статті описано реалізацію нового підходу до прихованого збирання інформації про діяльність зловмисника на системі-приманці. Також вирішено одну з найголовніших проблем безпечного застосування систем-приманок — стійкість до розкриття та блокування засобів прихованого моніторингу. Надалі планується розширення механізму інтерпретації подій для віртуалізованих ОС на базі Microsoft Windows.

Ще одним важливим доробком, над яким нами ведеться робота для майбутнього ефективного використання систем-приманок, є можливість контролю файлових ресурсів та оперативної пам'яті на проникнення вірусів та іншого шкідливого коду.

У запропонованому рішенні вирішено ще одну важливу проблему систем-приманок – можливість розгортання ферми віртуальних приманок [7] на одному сервері завдяки віртуалізації ОС та централізованому моніторингу віртуальної мережі.

Важливим моментом для подальшого застосування механізму є правильний вибір даних, який ми хотіли б отримувати (оскільки зазвичай журнали реєстрації всіх системних подій є надзвичайно громіздкими), а тому необхідно формалізувати механізм моніторингу лише найнеобхідніших фрагментів атаки.

Реалізоване програмне забезпечення дало можливість виконати дослідження, які свідчать про ефективність використання програмного забезпечення для повноцінного і відмовостійкого моніторингу ОС, для виявлення шкідливого програмного забезпечення, обліку активності програмного забезпечення, зокрема аномальної поведінки (виконання підміни системних викликів, підміна модулів ядра тощо), аналізу поширення вірусного та троянського програмного забезпечення.

1. Дудикевич В.Б., Піскозуб А.З., Тимошик Н.П., Тимошик Р.П. Новітні підходи прихованого збору інформації про діяльність зловмисника на системі-приманці // Науково-технічний журнал "Захист інформації" № 4, 2008. 2. Дудикевич В.Б., Піскозуб А.З., Тимошик Н.П., Тимошик Р.П., Дуткевич Т.В. Методи та засоби аналізу системи-приманки в процесі зламу // Вісник поліграфічного інституту. 2008 р. 3. Garfinkel T. and Rosenblum M. A Virtual Machine Introspection Based Architecture for Intrusion Detection. Proc. of the 2003 Network and Distributed System Security Symposium, Feb. 2003. 4. Bellard F. QEMU, a Fast and Portable Dynamic Translator. Proc. of USENIX Annual Technical Conference 2005 (FREENIX Track), July 2005. 5. Love R. Linux Kernel Development. Novell Press, 2nd edition, 2005. 6. Petroni N., Fraser T., Walters A. and Arbaugh W. An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data. Proc. of the 15th USENIX Security Symposium, Aug. 2006. 7. Asrigo K., Litty L., and Lie D. Using VMM-based sensors to monitor honeypots. In Proceedings of the 2nd ACM/USENIX International Conference on Virtual Execution Environments,2006.