Facultad de Ingeniería
Universidad de Concepción

Code-reuse attacks and defenses

Juan Rodrigo Anabalón R.

Estudiante de doctorado en Universidad del Cauca – Colombia

Doctorado en Ciencias de la Electrónica, Área Computación

Facultad de Ingeniería Electrónica y Telecomunicaciones

Universidad del Cauca

Popayán – Cauca - Colombia

Who I am

Juan Rodrigo Anabalón R.

Es director y fundador de MonkeysLab Presidente del Information Systems Security Association – Chile (issa-chile.cl). Recibió una Licenciatura en Ciencias de la Ingeniería y el título de Ingeniero de Ejecución en Informática en Universidad de las Américas y el grado de Magíster en Seguridad, Peritaje y Auditoría en Procesos Informáticos en la Universidad de Santiago de Chile, es certificado en Homeland Security and Cybersecurity en University of Colorado y estudiante de Doctorado en Ciencias de la Electrónica en Universidad del Cauca, Colombia. Sus áreas investigación están vinculados a ciberseguridad e infraestructura crítica, explotación de software, seguridad ofensiva y ciberoperaciones. Escribe regularmente sobre ciberseguridad en http://deoxyt2.livejournal.com.

Agenda

- Breve reseña de explotación de sistemas
- Corrupción de memoria clásica
- Ataques de reutilización de códigos
- Áreas de interés en investigación
- Conclusiones

Grandes aspectos de seguridad

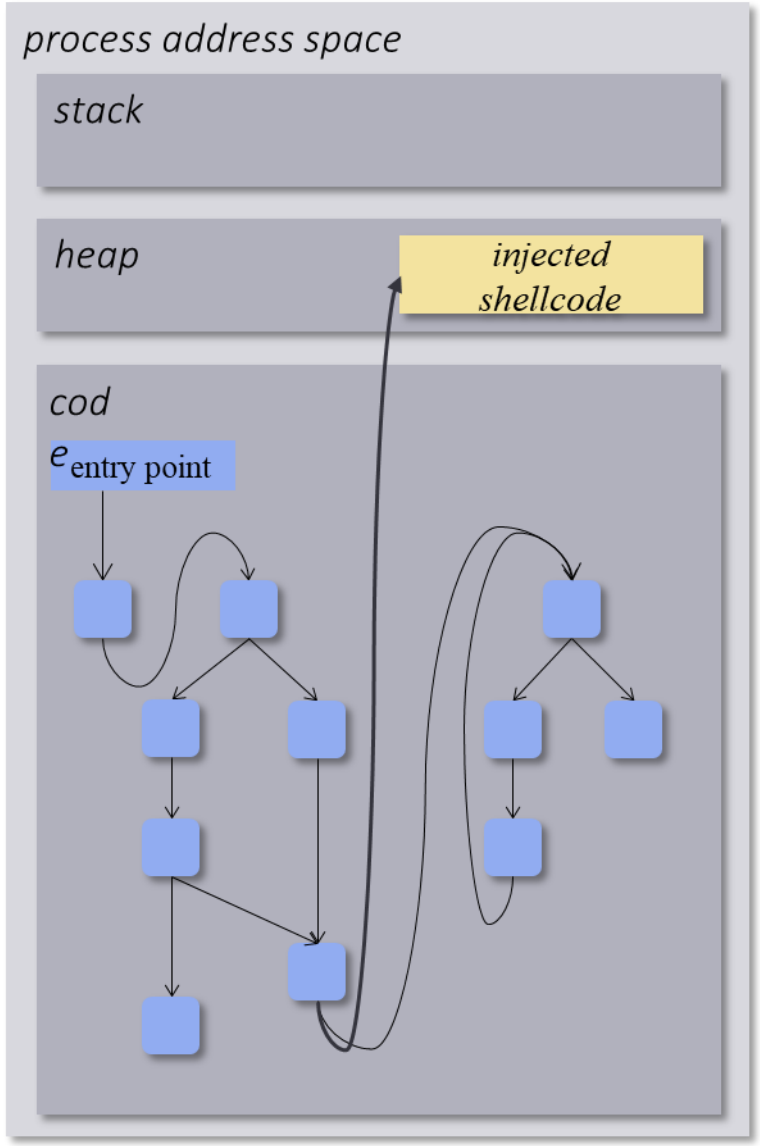
- Atacar
- Proteger
- Auditar

Origen

- La familia C de lenguajes de programación (C, C++, C#) es uno de los géneros de lenguajes de programación más utilizados, si no el más utilizado.
- La administración de memoria es responsabilidad del programador.

Objetivo del ataque

- Efectuar alguna intención maliciosa secuestrando el flujo de control del programa
- Históricamente, logrado a través de la inyección de código



Detecting Code Reuse Attacks with a Model of Conformant Program Execution. Emily R. Jacobson, Andrew R. Bernat, William R. Williams, Barton P. Miller

Como funciona syscall

- Shellcode es un término que, desde su origen, específica un exploit utilizado para generar un shell root.
- Syscalls permite acceder a funciones de SO: obtener entrada, producir salida, salir de un proceso y ejecutar un archivo binario.
- También permite acceder directamente al kernel.
  - Biblioteca libc: Muy utilizada para ataques return-into-libc, donde el atacante sobrescribe la dirección de retorno de una función vulnerable con la dirección de cualquier función de la biblioteca libc.
  - llamada sys: Se puede hacer una llamada a sys directamente con assembly. cargando los argumentos adecuados en registros y, a continuación, llamando a una interrupción de software.

Syscall y Assembly

- Un computador no hace distinción entre instrucciones y datos.
- Esto hace posible la explotación de sistemas.
- En IA32 los registros de uso general para operaciones matemáticas incluyen registros EAX, EBX y ECX, y se utilizan para almacenar datos y direcciones, direcciones de desplazamiento, realizar funciones de recuento y muchas otras cosas.
- El Extended Stack Pointer (ESP) señala la dirección de memoria donde tendrá lugar la siguiente operación de stack. Para entender un desbordamiento de stack se debe comprender cómo se utiliza ESP con instrucciones de assembly y el efecto que tiene en los datos del stack.
- Extended Instruction Pointer (EIP) contiene la dirección de la siguiente instrucción de máquina que se va a ejecutar.
- Si desea controlar la ruta de ejecución de un programa, es importante tener la capacidad de acceder y cambiar el valor almacenado en el registro EIP. Por último,
- Extended Flags (EFLAGS) comprende muchos registros de un solo bit que se utilizan para almacenar los resultados de varias pruebas realizadas por el procesador.

Syscall

- Las llamadas del sistema en Linux se realizan a través de interrupciones de software y se llaman con la instrucción int 0x80
- El proceso funciona de la siguiente manera:
  - El número de llamada sys específico se carga en EAX.
  - Los argumentos de la función syscall se colocan en otros registros.
  - Se ejecuta la instrucción int0x80.
  - La CPU cambia al modo kernel.
  - Se ejecuta la función syscall.

```
#include<stdlib.h>
int main() {
        exit(0);
}
```

$gcc –static –o exit exit.c

jar@debian:~$ gdb exit

GDB

Una instrucción que carga el argumento a nuestra llamada syscall en EBX y tiene el valor cero.

```
0x0806c553 <+0>:     mov    0x4(%esp),%ebx
```

El valor de la llamada de sistema se almacena en EAX en líneas 0x0806c557 <+4> y 0x0806c563 <+16>:

```
0x0806c557 <+4>:     mov    $0xfc,%eax
0x0806c563 <+16>:    mov    $0x1,%eax
```

Finalmente, tenemos la instrucción int 0x80, que cambia la CPU al modo kernel que es el que ejecuta realmente el syscalls:

```
0x0806c568 <+21>:    int    $0x80
```

Todo esto es lo que hace assembly cuando ejecutamos una simple llamada exit() a syscall.

```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from exit...(no debugging symbols found)...done.
(gdb) disas _exit
Dump of assembler code for function _exit:
0x0806c553 <+0>:     mov    0x4(%esp),%ebx
0x0806c557 <+4>:     mov    $0xfc,%eax
0x0806c55c <+9>:     call   *%gs:0x10
0x0806c563 <+16>:    mov    $0x1,%eax
0x0806c568 <+21>:    int    $0x80
0x0806c56a <+23>:    hlt
End of assembler dump.
(gdb)
```

Stack Overflows

- 1996 "Smashing the Stack for Fun and Profit" de Aleph One.
- La matriz, denominada array, tiene una longitud de cinco elementos[0-4].
- El compilador gcc no alerta errores, pero cuando ejecutamos este código, obtenemos resultados inesperados.

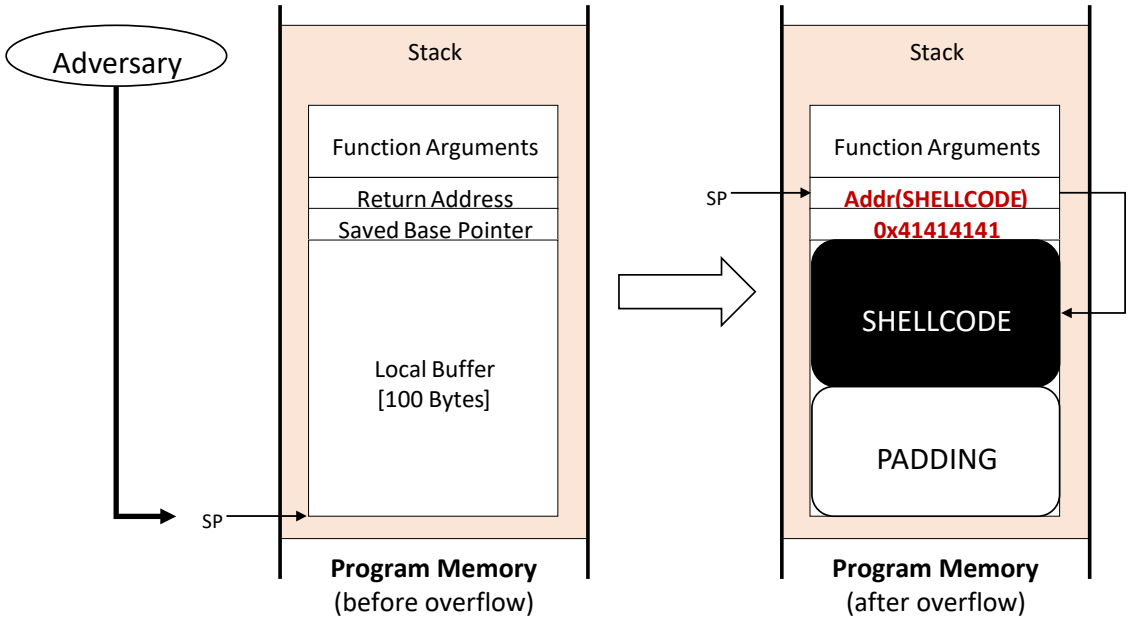
```
$ cc buffer.c
$ ./a.out
134513712
```

```c
#include <stdio.h>
#include <string.h>

int main ()
{
    int array[5] = {1, 2, 3, 4, 5};

    printf("%d\n", array[5] );
}
```

Memory layout of code injection attacks



Memory Corruption

- Stack Buffer Overflow
- Heap corruption
- Heap spraying
- Use-After-Free
- Double-Free

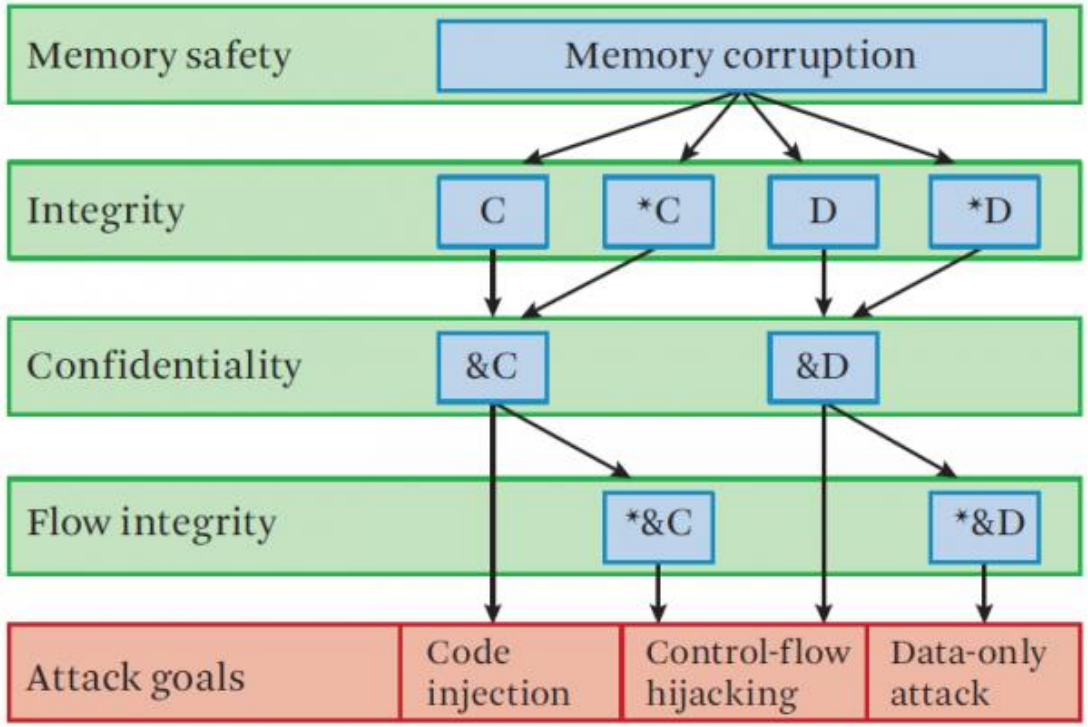


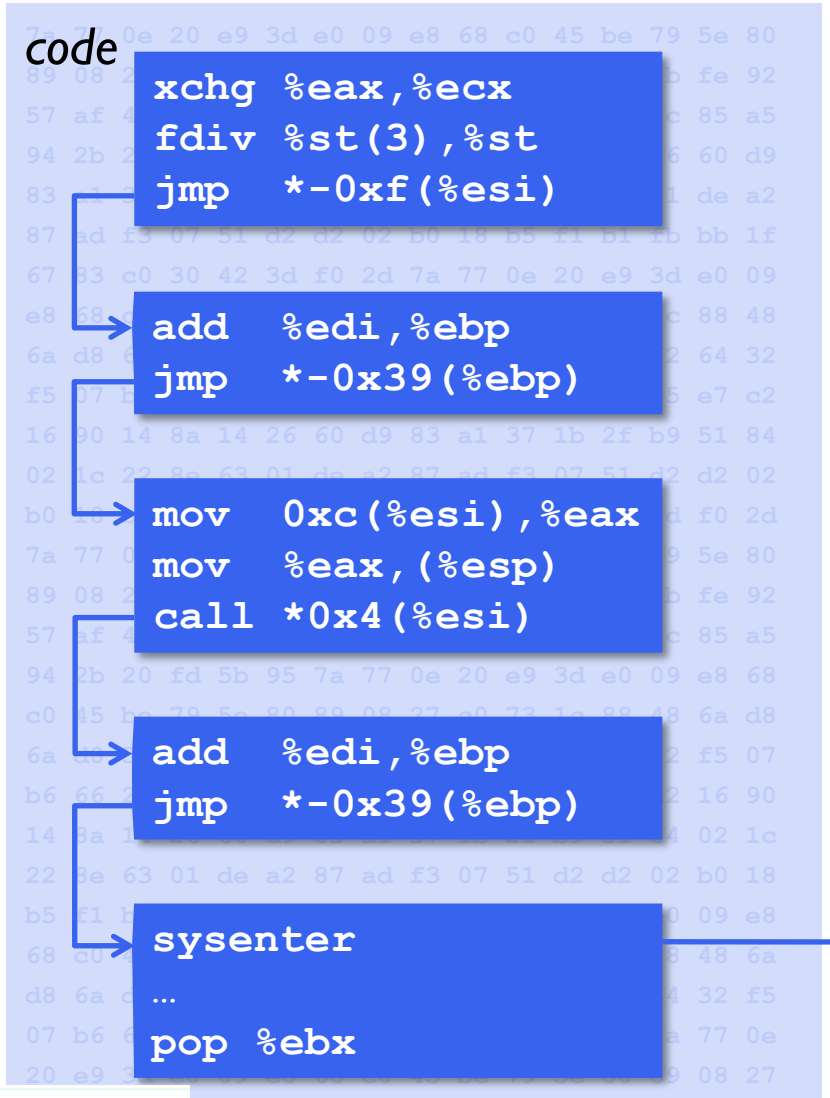
Figure 1 Superficie de ataque, que muestra las rutas de ataque desde la infracción de seguridad de la memoria inicial hasta la inyección de código, control-flow hijacking, y data-only attacks. En la figura, C muestra código, D muestra datos, asterisco (*) marca punteros, ampersand (&) marca direcciones y *& marca una operación de desreferencia, por ejemplo, *&C nos dice que se está desrreferenciando un puntero de código.

The Continuing Arms Race. Code-Reuse Attacks and Defenses. Per Larsen, Ahmad-Reza Sadeghi.

Code Reuse Attacks

- Los ataques de reutilización de código (Code-Reuse Attacks) son vulnerabilidades de seguridad que permiten a los atacantes ejecutar código arbitrario en una máquina comprometida, entre los ejemplos de este tipo de ataques se pueden mencionar los return-oriented programing (ROP) y los jump-oriented programming (JOP) que reutilizan fragmentos del código de una librería, evitando así la necesidad de inyección explícita de código en el stack.

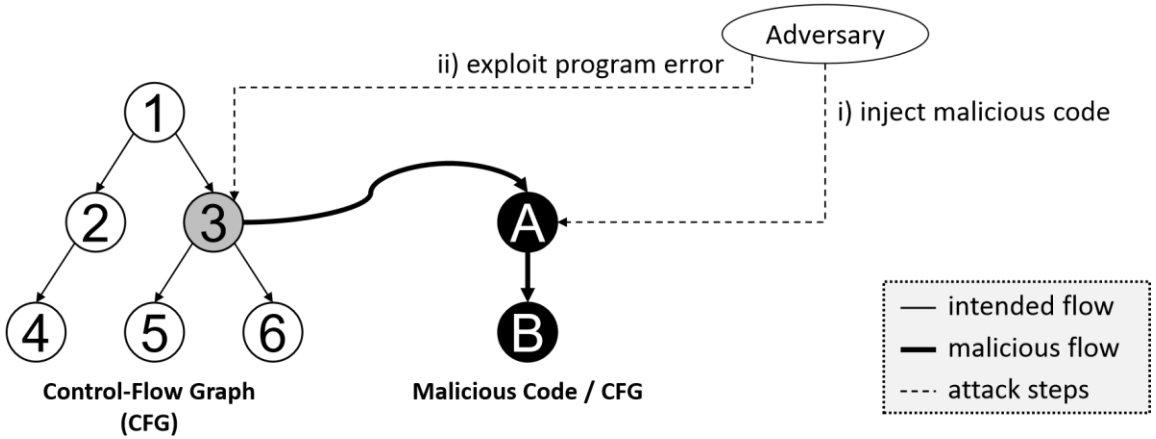
Anatomía CRA

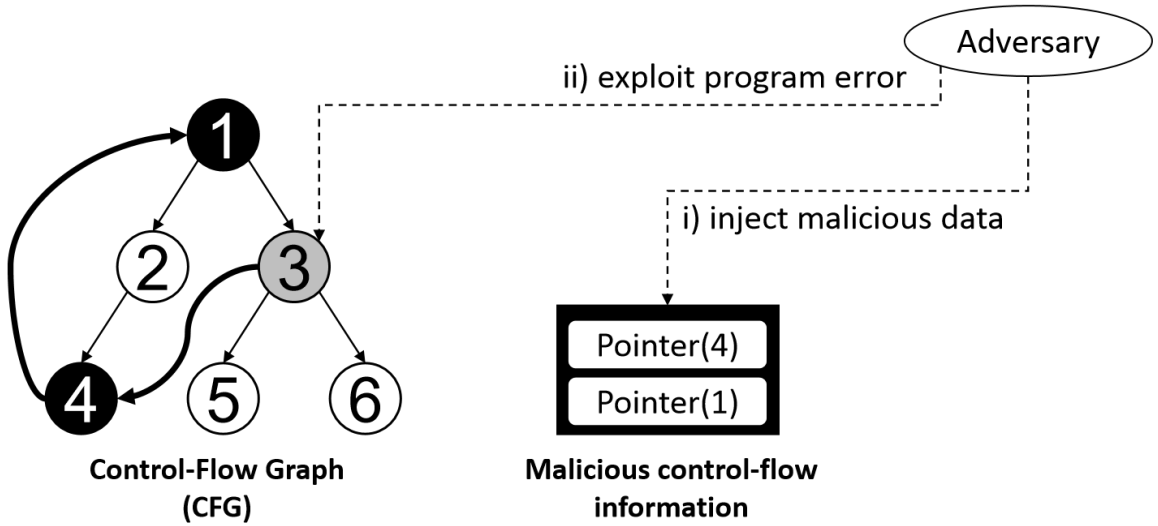


- Seleccione gadgets desde el espacio de direcciones del proceso
- Encadenar gadgets junto con control indirecto de flujo instrucciones de salto indirecto
- instrucciones de salto indirecto
  - "programación orientada al salto" (JOP)
- Por lo general, un ataque corto con el objetivo de evadir del confinamiento entorno W ⊕X

Detecting Code Reuse Attacks with a Model of Conformant Program Execution. Emily R. Jacobson, Andrew R. Bernat, William R. Williams, Barton P. Miller

Code injection attacks vs Code-reuse attacks





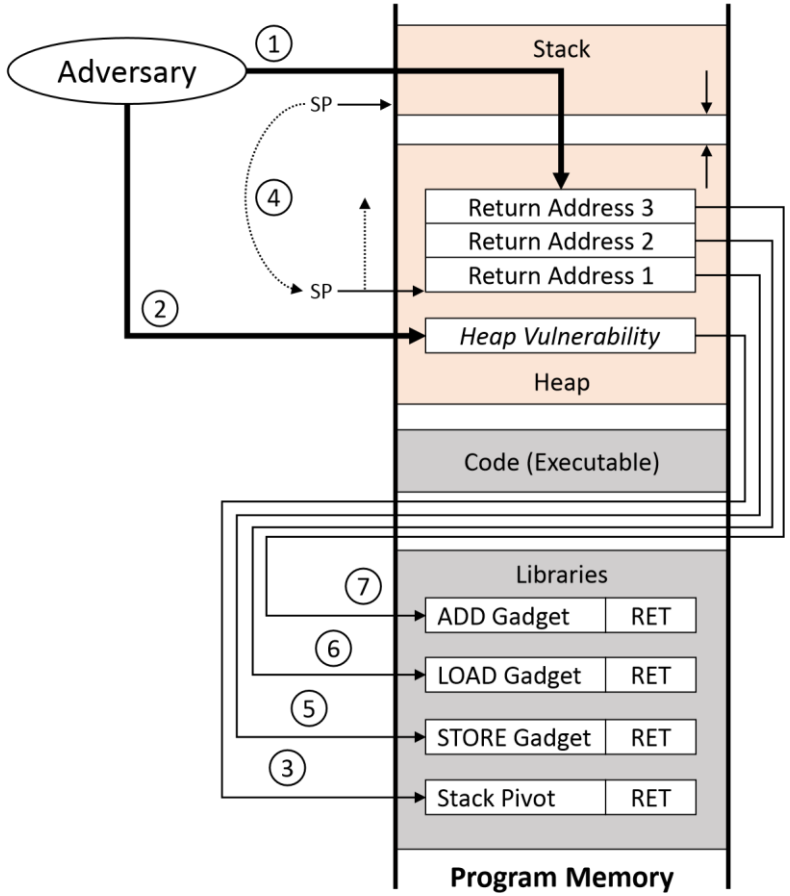
Lucas Vincenzo Davi:
Code-Reuse Attacks and Defenses, © April 2015

Vista General CRA

- Return-to-libc:
  - En un ataque return-into-libc, el atacante sobrescribe la dirección de retorno de una función vulnerable con la dirección de cualquier función de la biblioteca libc. Por ejemplo, el atacante podría sobrescribir con la dirección de una función "system", por la cual se puede abrir una Shell.
  - Se debe controlar EIP se puede poner cualquier dirección de destino, como por ejemplo libc.
- return-to-user (ret2usr
  - return-to-user (ret2usr), una variante a nivel de kernel de return-into-libc.
- Return-Oriented Programming (ROP)
  - Return-Oriented Programming (ROP) es una forma específica de ataque return-into-libc en el que el atacante ejecuta secciones de código dispersas en el espacio de direcciones del proceso vinculándolas con instrucciones de transferencia de control indirecto, normalmente la instrucción ret.
- Jump-Oriented Programming (JOP)
  - Jump-oriented programming (JOP) es una técnica que, en vez de usar instrucciones ret para enlazar gadgets, utiliza una llamada indirecta o salto para subvertir el control de flujo, lo que permite saltarse las innumerables defensas contra ataques (ROP).
- Counterfeit Object-oriented Programming (COOP)
  - Muchas defensas CRA (tales como, CPS, T-VIP, vfGuard y VTint) no consideran la semántica de C++ orientada a objetos con precisión pueden ser omitidas. Induce el comportamiento malicioso del programa al invocar cadenas de funciones virtuales de C ++ existentes en un programa a través de llamadas existentes.
- String-Oriented Programming (SOP)
  - Explota las regiones de memoria estática en aplicaciones con ASLR habilitado.

Return-oriented programming

- Propuesto por Hovav Shacham “The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)”
- ROP es una forma específica de ataque return-into-libc
- Hace uso de secciones de código dispersas en el espacio de direcciones vinculándolas con instrucciones de transferencia de control indirecto, normalmente la instrucción "ret".
- Permite la ejecución arbitraria de código para explotar vulnerabilidades de desbordamiento de búfer en el stack y en heap sin sobrescribir la dirección de retorno.
- Puede evadir defensas como Protección Data Execution Prevention (DEP)



Principio básico de los ataques de programación orientados al retorno

Jump-oriented programming (JOP)

- Propuesto por Tyler Bletsch “Jump-oriented programming: a new class of code-reuse attack”

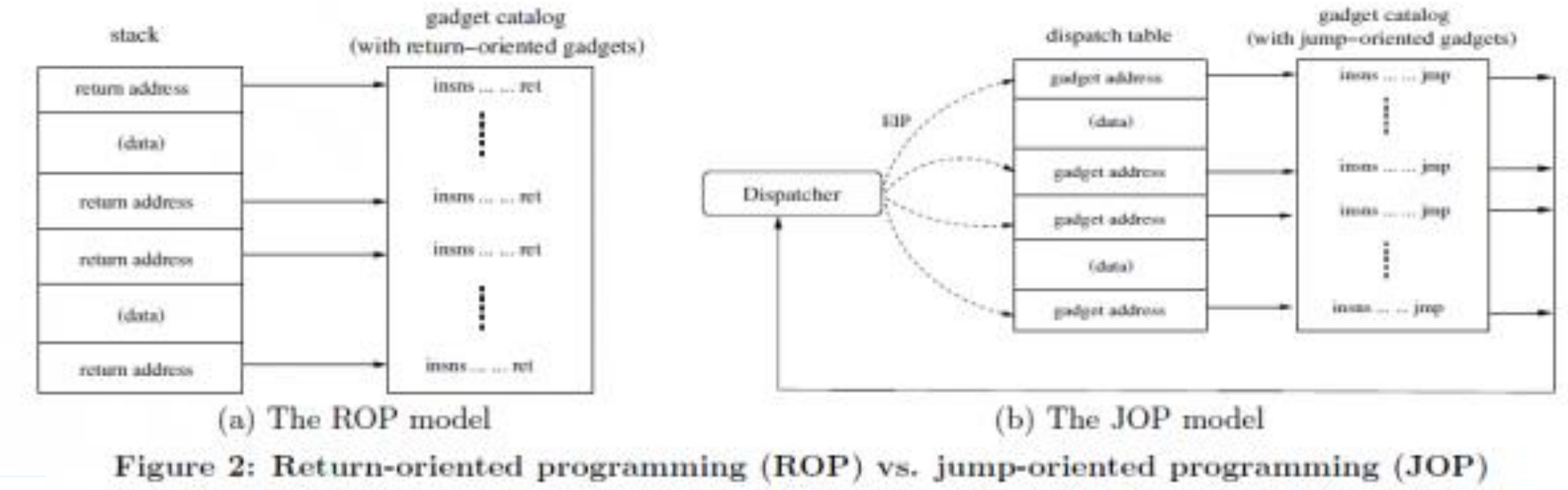


Figure 2: Return-oriented programming (ROP) vs. jump-oriented programming (JOP)

T. Bletsch, X. Jiang, V. W. Freeh, y Z. Liang, «Jump-oriented programming: a new class of code-reuse attack», en *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, New York, NY, USA, mar. 2011, pp. 30-40. doi: 10.1145/1966913.1966919.

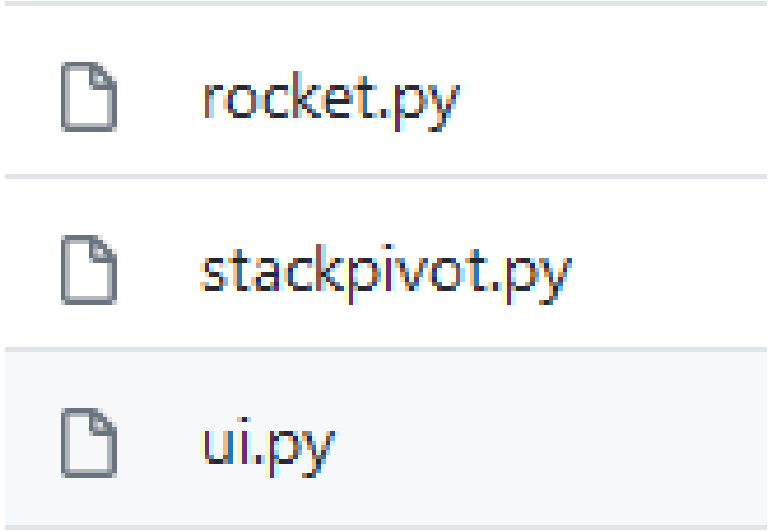
Jump-oriented programming

- Existen tres variedades distintas de realizar JOP:
  - Bring Your Own Pop Jump (BYOPJ), introducido por Checkoway y Shacham[16].
  - Table dispacher: Introducido por Bletsch en 2011 [13], y
  - Control Gadget y "combinational gadgets": Una variación en el paradigma de dispatcher gadget de Bletsch[17].

JOP Rocket

- JOP ROCKET es una herramienta diseñada para ayudar a facilitar el descubrimiento de gadgets JOP en un entorno Windows x86. Esta herramienta fue lanzada en DEF CON 27, donde fue objeto de una charla del Dr. Bramwell Brizendine y el Dr. Josh Stroschein. Una importante actualización está en desarrollo se publicó en septiembre de 2020, con actualizaciones menores previstas en un futuro próximo.

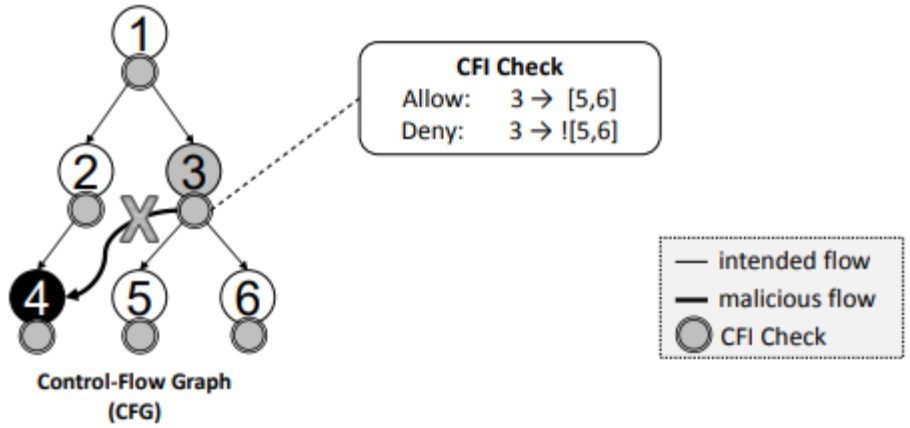
https://github.com/Bw3ll/JOP_ROCKET/



String-Oriented Programming (SOP)

- Propuesto por Mathias Payer y Thomas R. Gross. “String Oriented Programming: When ASLR is not Enough”.
- Explota las regiones de memoria estática en aplicaciones con ASLR habilitado.
- SOP utiliza un error de *string format* para explotar aplicaciones que están protegidas por una combinación de ASLR, DEP y stack canaries.
  - String format inyectan datos en aplicaciones en ejecución.
  - Canaries es una técnica de protección que establece variables con valores especiales que se colocan junto a los búferes en el stack o en heap y sus valores se validan para proteger el sistema contra buffer overflows.
  - Address Space Layout Randomization (ASLR) es una protección probabilística que asigna al azar las ubicaciones del código, stack, heap, y otros datos.

Control-flow integrity (CFI)



- CFI ofrece una defensa genérica contra los ataques de reutilización de código al validar la integridad del flujo de control de un programa basado en un gráfico de flujo de control (CFG) predefinido en tiempo de ejecución.
- Antes de la ejecución del programa, se debe identificar el CFG de la aplicación.
- A continuación, se emite una comprobación de CFI en la instrucción de salida de cada nodo CFG.
- Estas comprobaciones de CFI se ejecutan en tiempo de ejecución para evitar cualquier intento del adversario de secuestrar el flujo de control del programa.
- Por ejemplo, la comprobación de CFI en el nodo n3 valida que la instrucción de salida solo se dirige a n5 o n6. Si el adversario tiene como objetivo redirigir la ejecución a n4, CFI terminará inmediatamente la ejecución del programa.
- MoCFI, CCFIR, Bin-CFI, kBouncer, ROPecker, O-CFI, Virtual-Table Verification (VTV) y Indirect Function-Call Checks (IFCC), SafeDispatch, VfGuard, VTint, PathArmor, TypeArmor, SafeDispatch, VTV, VTrust y VTI.

Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity: principles, implementations and applications. En: CCS (2005)

Importancia de la investigación

- La ejecución de JOP es compleja y desalentadora.
- Desarrollar nuevas capacidades de explotación JOP puede facilitar la tarea de investigación de vulnerabilidades.
- Permitir a los investigadores descubrir vulnerabilidades desconocidas y así permitir la construcción de contramedidas antes que los actores maliciosos las exploten.
- CFG proporcionará defensa contra JOP y es necesario investigar forms de explotación antes de los actores maliciosos.
- Las actuales herramientas de explotación ROP/JOP están basadas en 32 bits y para Windows 7, plataforma que ya está en retirada.

Áreas interesantes de investigación

- Desarrollar una herramienta que permita la explotación JOP en Arquitectura de 64 bits (AMD/Intel/ARM).
- Desarrollar una herramienta que permita la explotación JOP en Arquitectura de 64 bits (AMD/Intel/ARM).
- Superar Python y desarrollar herramientas que permitan trabajar a bajo nivel nativamente.
  - ¿Alguien dijo Rust?
  - ¿JuliaLang?
- Se han propuesto defensas para JOP, ¿podemos evadirlas?.
- Metaheurísticas
- Lenguajes de programación cuánticos (QCL, Q#, QMASM etc.)

A. Salas-Fernández, B. Crawford, R. Soto, y S. Misra, «Metaheuristic Techniques in Attack and Defense Strategies for Cybersecurity: A Systematic Review», Artificial Intelligence for Cyber Security: Methods, Issues and Possible Horizons or Opportunities, pp. 449-467, 2021, doi: 10.1007/978-3-030-72236-4_18.

Conclusiones

- Ataques clásicos están superados y cada vez son más difíciles de realizar.
- CRA representan una nueva amenaza y se están desarrollando nuevos tipos de ataques de reutilización de código.
- CRA son complejos de realizar y no se cuenta con herramientas o frameworks para facilitar el trabajo de los investigadores.
- Es necesaria la investigación de este tipo de ataques en otras arquitecturas diferentes a x32.

Referencias (1/3)

[1] Y. Ruan, S. Kalyanasundaram, y X. Zou, «Survey of return-oriented programming defense mechanisms», Secur. Commun. Netw., vol. 9, n.o 10, pp. 1247-1265, jul. 2016, doi: 10.1002/sec.1406.

[2] L. Davi, A.-R. Sadeghi, y M. Winandy, «Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks», en Proceedings of the 2009 ACM workshop on Scalable trusted computing, New York, NY, USA, nov. 2009, pp. 49–54, doi: 10.1145/1655108.1655117.

[3] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, y L. Xie, «DROP: Detecting Return-Oriented Programming Malicious Code», en Information Systems Security, Berlin, Heidelberg, 2009, pp. 163-177, doi: 10.1007/978-3-642-10772-6_13.

[4] J. Li, Z. Wang, X. Jiang, M. Grace, y S. Bahram, «Defeating return-oriented rootkits with "Return-Less" kernels», en Proceedings of the 5th European conference on Computer systems, New York, NY, USA, abr. 2010, pp. 195–208, doi: 10.1145/1755913.1755934.

[5] M. Kayaalp, M. Ozsoy, N. A. Ghazaleh, y D. Ponomarev, «Efficiently Securing Systems from Code Reuse Attacks», IEEE Trans. Comput., vol. 63, n.o 5, Art. n.o 5, may 2014, doi: 10.1109/TC.2012.269.

[6] A. One, .:: «Phrack Magazine ::.», Phrack magazine, vol. 49, jul. 16, 2020.

[7] H. Meer, «Memory Corruption AttacksThe (almost) Complete History», presentado en Backhat 2010, Las Vegas, NV, jun. 2010, [En línea]. Disponible en: https://media.blackhat.com/bh-us-10/whitepapers/Meer/BlackHat-USA-2010-Meer-History-of-Memory-Corruption-Attacks-wp.pdf.

[8] F.-H. Hsu, C.-H. Huang, C.-H. Hsu, C.-W. Ou, L.-H. Chen, y P.-C. Chiu, «HSP: A solution against heap sprays», J. Syst. Softw., vol. 83, n.o 11, pp. 2227-2236, nov. 2010, doi: 10.1016/j.jss.2010.06.045.

[9] «CWE - CWE-415: Double Free (4.1)». https://cwe.mitre.org/data/definitions/415.html (accedido jul. 26, 2020).

[10] M. Mouzarani, B. Sadeghiyan, y M. Zolfaghari, «A smart fuzzing method for detecting heap-based vulnerabilities in executable codes», Secur. Commun. Netw., vol. 9, n.o 18, pp. 5098-5115, dic. 2016, doi: 10.1002/sec.1681.

[11] A. (Solar D. Peslyak, «Bugtraq: Getting around non-executable stack (and fix)». https://seclists.org/bugtraq/1997/Aug/63 (accedido jul. 26, 2020).

Referencias (2/3)

[1] Y. Ruan, S. Kalyanasundaram, y X. Zou, «Survey of return-oriented programming defense mechanisms», Secur. Commun. Netw., vol. 9, n.o 10, pp. 1247-1265, jul. 2016, doi: 10.1002/sec.1406.

[2] L. Davi, A.-R. Sadeghi, y M. Winandy, «Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks», en Proceedings of the 2009 ACM workshop on Scalable trusted computing, New York, NY, USA, nov. 2009, pp. 49–54, doi: 10.1145/1655108.1655117.

[3] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, y L. Xie, «DROP: Detecting Return-Oriented Programming Malicious Code», en Information Systems Security, Berlin, Heidelberg, 2009, pp. 163-177, doi: 10.1007/978-3-642-10772-6_13.

[4] J. Li, Z. Wang, X. Jiang, M. Grace, y S. Bahram, «Defeating return-oriented rootkits with "Return-Less" kernels», en Proceedings of the 5th European conference on Computer systems, New York, NY, USA, abr. 2010, pp. 195–208, doi: 10.1145/1755913.1755934.

[5] M. Kayaalp, M. Ozsoy, N. A. Ghazaleh, y D. Ponomarev, «Efficiently Securing Systems from Code Reuse Attacks», IEEE Trans. Comput., vol. 63, n.o 5, Art. n.o 5, may 2014, doi: 10.1109/TC.2012.269.

[6] A. One, .:: «Phrack Magazine ::.», Phrack magazine, vol. 49, jul. 16, 2020.

[7] H. Meer, «Memory Corruption AttacksThe (almost) Complete History», presentado en Backhat 2010, Las Vegas, NV, jun. 2010, [En línea]. Disponible en: https://media.blackhat.com/bh-us-10/whitepapers/Meer/BlackHat-USA-2010-Meer-History-of-Memory-Corruption-Attacks-wp.pdf.

[8] F.-H. Hsu, C.-H. Huang, C.-H. Hsu, C.-W. Ou, L.-H. Chen, y P.-C. Chiu, «HSP: A solution against heap sprays», J. Syst. Softw., vol. 83, n.o 11, pp. 2227-2236, nov. 2010, doi: 10.1016/j.jss.2010.06.045.

[9] «CWE - CWE-415: Double Free (4.1)». https://cwe.mitre.org/data/definitions/415.html (accedido jul. 26, 2020).

[10] M. Mouzarani, B. Sadeghiyan, y M. Zolfaghari, «A smart fuzzing method for detecting heap-based vulnerabilities in executable codes», Secur. Commun. Netw., vol. 9, n.o 18, pp. 5098-5115, dic. 2016, doi: 10.1002/sec.1681.

[11] A. (Solar D. Peslyak, «Bugtraq: Getting around non-executable stack (and fix)». https://seclists.org/bugtraq/1997/Aug/63 (accedido jul. 26, 2020).

[12] «The geometry of innocent flesh on the bone | Proceedings of the 14th ACM conference on Computer and c https://dl.acm.org/doi/10.1145/1315245.1315313 (accedido jul. 26, 2020).

Referencias (3/3)

[13] T. Bletsch, X. Jiang, V. W. Freeh, y Z. Liang, «Jump-oriented programming: a new class of code-reuse attack», en Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, Hong Kong, China, mar. 2011, pp. 30–40, doi: 10.1145/1966913.1966919.

[14] L. Erdődi, «Attacking x86 windows binaries by jump oriented programming», en 2013 IEEE 17th International Conference on Intelligent Engineering Systems (INES), jun. 2013, pp. 333-338, doi: 10.1109/INES.2013.6632837.

[15] R. Qiao, M. Zhang, y R. Sekar, «A Principled Approach for ROP Defense», en Proceedings of the 31st Annual Computer Security Applications Conference, Los Angeles, CA, USA, dic. 2015, pp. 101–110, doi: 10.1145/2818000.2818021.

[16] S. Checkoway y H. Shacham, «Escape from return-oriented programming: Returnoriented programming without returns (on the x86», 2010.

[17] P. Chen, X. Xing, B. Mao, L. Xie, X. Shen, y X. Yin, «Automatic construction of jump-oriented programming shellcode (on the x86)», en Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, New York, NY, USA, mar. 2011, pp. 20–29, doi: 10.1145/1966913.1966918.

[18] B. Tyler, «Code-Reuse Attacks: New Frontiers and Defenses», dissertation, North Carolina State University, North Carolina, 2011.

[19] O. Whitehouse, «"An Analysis of Address Space Layout Randomization on Windows Vista», feb. 2007, p. 16.

[20] «VTint: Protecting Virtual Function Tables' Integrity – NDSS Symposium». https://www.ndss-symposium.org/ndss2015/ndss-2015-programme/vtint-protecting-virtual-function-tables-integrity/ (accedido ago. 13, 2020).

[21] J. Brandon, «jdbrandon/ControlFlowIntegrity», GitHub. https://github.com/jdbrandon/ControlFlowIntegrity (accedido ago. 09, 2020).

[22] A. R. Hevner, S. T. March, J. Park, y S. Ram, «Design science in information systems research», MIS Q., vol. 28, n.o 1, pp. 75–105, mar. 2004.

[23] B. Schneier, «"People, Process, and Technology" - Schneier on Security». https://www.schneier.com/blog/archives/2013/01/people_process.html (accedido ago. 09, 2020).