วิชา 204554 การเข้ารหัสลับและความปลอดภัยของข้อมูล

การบ้านที่ 3 (นำมาจาก https://www.handsonsecurity.net/resources.html)

ตอบคำถามต่อไปนี้ลงในไฟล์ studentID_firstname_hw3.pdf โดย studentID และ firstname คือรหัสและชื่อแรกของตัวนิสิตตามลำดับ <u>คำตอบที่จะได้คะแนนจะต้องมีเหตุผลกำกับด้วยเสมอ</u>

1. บอกความแตกต่างระหว่าง environment variable กับ shell variable
2. ใน Bash shell ถ้าเรารันคำสั่ง `export foo=bar` จะทำให้ environment variable ของ process ปัจจุบันเปลี่ยนไปหรือไม่
3. จงบอกความแตกต่างในการพิมพ์ค่า environment variable โดยใช้คำสั่งในสองลักษณะต่อไปนี้

```
$ /usr/bin/env
$ /usr/bin/strings /proc/$$/environ
```

4. เมื่อเราใช้ execve() ในการรันโปรแกรมภายนอก xyz และเราให้ค่า argument ที่สามใน execv() เป็น NULL จงบอกจำนวนของ environment variable ทั้งหมดที่ process xyz มี
5. Bob บอกว่าเขาไม่เคยใช้งาน environment variable ในโปรแกรมของเขาเลย ดังนั้นเขาจึงไม่ต้องกังวลเรื่องความปลอดภัยที่เกี่ยวข้องกับ environment variable ในโปรแกรมของเขาแต่อย่างใด เขาคิดถูกหรือไม่
6. โปรแกรม abc เรียกใช้งานโปรแกรมภายนอก xyz โดยใช้ system() เมื่อเรารัน abc ผ่าน shell อธิบายว่าค่า PATH shell variable มีผลกระทบต่อ system() อย่างไรบ้าง
7. อธิบายว่าทำไมเราจึงควรใช้งาน secure-getenv() มากกว่า getenv()
8. โปรแกรม Set-UID ที่ได้สิทธิ root ต้องการรู้ว่าขณะนี้ตัวมันรันอยู่ที่ directory ไหน วิธีหนึ่งที่ทำได้คือการดูจากค่า PWD environment variable ที่ระบุ full path ของ directory ปัจจุบัน อีกวิธีหนึ่งคือใช้ฟังก์ชั่น getcwd() (ขอให้นิสิตค้นคว้าข้อมูลเกี่ยวกับฟังก์ชั่นนี้จากอินเตอร์เน็ต) อธิบายว่าเราควรจะเลือกใช้วิธีไหน เพราะเหตุใด
9. ในระบบปฏิบัติการ Linux ตัว dynamic linker จะทิ้ง environment variable หลายๆตัวไปถ้ามันต้องรันในลักษณะโปรแกรม Set-UID ตัวอย่างที่เราได้เรียนรู้มาคือ LD_PRELOAD และ LD_LIBRARY_PATH ขอให้นิสิตศึกษาคู่มือออนไลน์จากลิงค์ต่อไปนี้

   https://linux.die.net/man/8/ld-linux

   อธิบายว่าทำไม LD_AUDIT กับ LD_DEBUG_OUTPUT ก็เป็น environment variable ที่ถูกละทิ้งไปด้วย

10. มีสองวิธีหลักในการให้ user ทั่วไปทำงานที่ต้องใช้สิทธิ root วิธีแรกคือใช้กลไก Set-UID ส่วนอีกวิธีหนึ่งคือให้มี process ของ root รันอยู่เป็น background (root daemon) อธิบายว่าวิธีไหนปลอดภัยกว่ากัน เพราะเหตุใด

11. ตัวแปร i str ptr buf j และ y ในส่วนของโปรแกรมด้านล่างนี้ แต่ละตัวอยู่ในบริเวณของ memory ในส่วนใด (stack heap data หรือ code)

```
int i = 0;
void func(char *str)
{
   char *ptr = malloc(sizeof(int));
   char buf[1024];
   int j;
   static int y;
}
```

12. เขียนภาพ stack frame ของฟังก์ชั่นภาษาซีต่อไปนี้ ให้วาดรวม frame ของ caller ที่เรียกใช้งาน bof และ frame ที่ bof ทำตัวเป็น caller เรียกไปหา strcpy ด้วย

```
int bof(char *str)
{
   char buffer[24];
   strcpy(buffer,str);
   return 1;
}
```

13. เนื่องจาก stack ที่เราพิจารณา เติบโตจาก address สูงไปที่ address ต่ำ และ buffer ที่ตำแหน่ง index 0 เริ่มต้นที่ address ต่ำ เช่นนี้เราสามารถ overflow buffer และเขียนทับ return address ซึ่งอยู่ที่ตำแหน่ง address สูงได้ ดังนั้นถ้าเราจะแก้ปัญหา buffer overflow เพียงแค่เราปรับให้ stack เติบโตจาก address ต่ำไป address สูง ปัญหา buffer overflow น่าจะหมดไป เพราะ return address จะอยู่ที่ address ต่ำก่อนที่จะมีการจอง buffer ที่จะอยู่ที่ address สูงกว่า และการ overflow จะไปในทิศทางที่ไม่ได้มี return addrees อยู่ วิจารณ์และประเมินความสำเร็จจากข้อเสนอแนะนี้

14. พิจารณาส่วนของโปรแกรมด้านล่างที่มีปัญหา buffer overflow จริงหรือไม่ที่การ overflow เกิดขึ้นภายใน strcpy() ดังนั้นการ jump ไปหา malware ที่ฉีดเข้ามาขณะทำการ overflow buffer จึงเกิดณ จังหวะที่ strcpy() return ไม่ใช่จังหวะที่ foo() return

```
int foo(char *str)
{
    char buffer[100];

    strcpy(buffer, str);

    return 1;
}
```

15. จากส่วนของโปรแกรมในข้อที่ 14 ถ้ารู้ว่า address ของ buffer เริ่มต้นที่ 0xbffff180 (&buffer[0]) พิจารณาการ overflow เพื่อไป overwrite ค่า return address บน stack ด้วยค่า address ที่จะนำ shellcode ไปวางดังต่อไปนี้

```
buffer address : 0xbffff180
case 1 : long retAddr = 0xbffff250 -> Able to get shell access
case 2 : long retAddr = 0xbffff280 -> Able to get shell access
case 3 : long retAddr = 0xbffff300 -> Cannot get shell access
case 4 : long retAddr = 0xbffff310 -> Able to get shell access
case 5:  long retAddr = 0xbffff400 -> Cannot get shell access
```

จะเห็นได้ว่าบางค่าทำการโจมตีได้สำเร็จ ขณะที่บางค่าทำไม่สำเร็จ พยายามอธิบายว่าค่าที่ทำให้การโจมตีไม่สำเร็จมีคุณสมบัติอย่างไร (คำใบ้: ดูว่าค่าที่ “Cannot get shell access” มีความเหมือนกันอย่างไร และลองค้นคว้าว่าไบท์ 0x00 มีนัยยะสำหรับ strcpy() อย่างไร)

16. จากส่วนของโปรแกรมด้านล่างนี้ ถ้า address เริ่มต้นของ buffer ใน bof คือ 0xAABB0010 ขณะที่ return address จะถูกเก็บไว้ที่ address 0xAABB0050 ค่าที่จะ overwrite ไปที่ address นี้ จะเป็นค่าที่เราจะนำ shellcode ไปวางไว้ ค่าที่ต่ำสุดที่ทำให้การเรียก shellcode สำเร็จมีค่าเท่าไหร่ (คำใบ้: ค้นคว้าว่า 0x00 มีนัยยะสำหรับ strcpy() อย่างไรและเราพบ 0x00 ใน address บน stack ที่เก็บ buffer และ return address หรือไม่)

```
int bof(char *str)
{
   char buffer[24];
   strcpy(buffer,str);
   return 1;
}
```

17. ทำไมเราจึงเรียกโค้ดที่ฉีดเข้ามาในขณะทำการโจมตีโดยใช้ buffer overflow ว่า shellcode อธิบายหลักการสร้าง shellcode อย่างง่ายและยกตัวอย่างประกอบ (คำใบ้: อ่านหัวข้อ 4.7 จากบทความในลิงค์นี้: www.handsonsecurity.net/files/chapters/buffer_overflow.pdf)

18. ถ้าเราต้องการให้ “/bin/sh” ที่เรียกผ่าน system() ในการโจมตีแบบ return-to-libc ไม่เกิดการ drop privilege ในกรณีที่ EUID ไม่เท่ากับ RUID อธิบายแนวทางเพื่อให้การโจมตีนี้สำเร็จ ให้ว่าเราสามารถใช้งานไบท์ 0x00 ได้ เพราะเรา overflow ผ่านทาง memcpy() ไม่ใช่ strcpy() (คำใบ้: ลองศึกษากลไกและการใช้งาน setuid(0))

19. ในการโจมตีแบบ return-to-libc ถ้าการ jump ไม่ได้ไปที่จุดเริ่มต้นของ system() แต่ไปที่คำสั่งแรกหลังจาก prologue ของ system() เราจะต้องสร้าง input array ในลักษณะใดให้การโจมตีสำเร็จลงได้ prologue ของฟังก์ชั่นในระบบ X86 32-bit ประกอบไปด้วยคำสั่งต่อไปนี้

```
push %ebp
mov %esp, %ebp
sub $N, %esp
```

โดย $N คือค่าคงที่ที่หารด้วย 4 ลงตัว

20. ASLR ช่วยป้องกันการโจมตีแบบ return-to-libc ได้มากน้อยขนาดไหน

21. การโจมตีแบบ buffer overflow และแบบ format string ทำให้เกิดการเปลี่ยนแปลงค่า return address บน stack ได้เหมือนกัน อธิบายว่ากลไกที่แตกต่างกันในทั้ง 2 วิธีและวิเคราะห์ว่ากลไกแบบใดที่มีข้อจำกัดน้อยกว่า

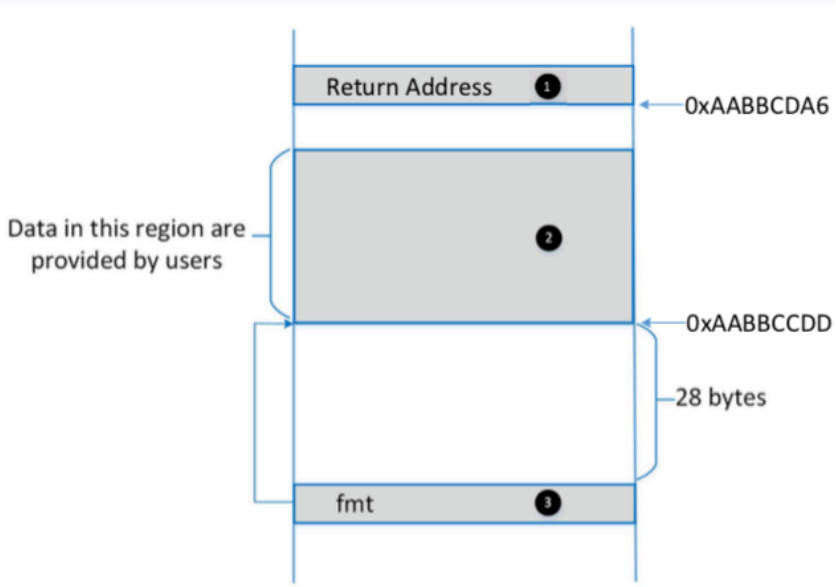
22. เราสามารถใช้เทคนิคในลักษณะเดียวกับ stack canary ในการป้องกันการโจมตีแบบ format string ได้หรือไม่

23. เมื่อคำสั่ง printf(fmt) รัน และข้อมูลบน stack จาก address ต่ำไป address สูงเรียงจากซ้ายไปขวาเป็นดังต่อไปนี้

```
0xAABBCCDD, 0xAABBDDFF, 0x22334455, 0x00000000, 0x99663322
```

ถ้าตัวซ้ายสุดคือ pointer ไปหา format string อธิบายว่าเราจะต้องใช้ % specifier กี่ตัว จึงจะทำให้เกิด segmentation error ได้แน่นอน 100%

24. ณ ตำแหน่งที่โปรแกรมรันมาถึงคำสั่ง printf(fmt); เพื่อพิมพ์ค่าที่รับมาจาก user ใน fmt buffer สภาวะของ stack เป็นดังภาพด้านล่าง



อธิบายว่าค่าที่รับมาจาก user จะต้องมีลักษณะเป็นอย่างไรจึงจะทำให้โปรแกรมหันมารันไวรัสที่ user ใส่เข้ามาที่ fmt buffer ได้ (ระบุค่าที่ใส่ใน address ที่เกี่ยวข้องเพื่อให้เกิดการรันไวรัสที่ฉีดเข้ามาได้)

25. วิเคราะห์ว่าการป้องกันต่อไปนี้มีผลช่วยป้องกันการโจมตีแบบ format string ได้มากน้อยแค่ไหน
    - ใช้กลไกคอมไพเลอร์ช่วยเตือนเวลาจำนวน argument ใน printf ไม่เท่ากับจำนวน specifier ใน format string
    - ใช้กลไก hardware/OS ทำให้ stack เป็นบริเวณ non-executable

การส่งงาน:

- **ส่งไฟล์ studentID_firstname_hw3.pdf ไปที่ Google Classroom ของวิชาภายในเวลาที่กำหนด**
- **ถ้าใครไม่ตั้งชื่อไฟล์ตามที่กำหนดหรือไม่ได้ส่งเป็นไฟล์แบบ pdf จะถูกหักคะแนนลงอีก 10% ของคะแนนที่ได้รับ**