GP1: BUENAS PRÁCTICAS DE PROGRAMACIÓN

Fundamentos de la seguridad en el software y en los componentes

Universidad de Alcalá

KEVIN KAREL VAN LIEBERGEN
JORGE LAFUENTE

ACTIVIDADES

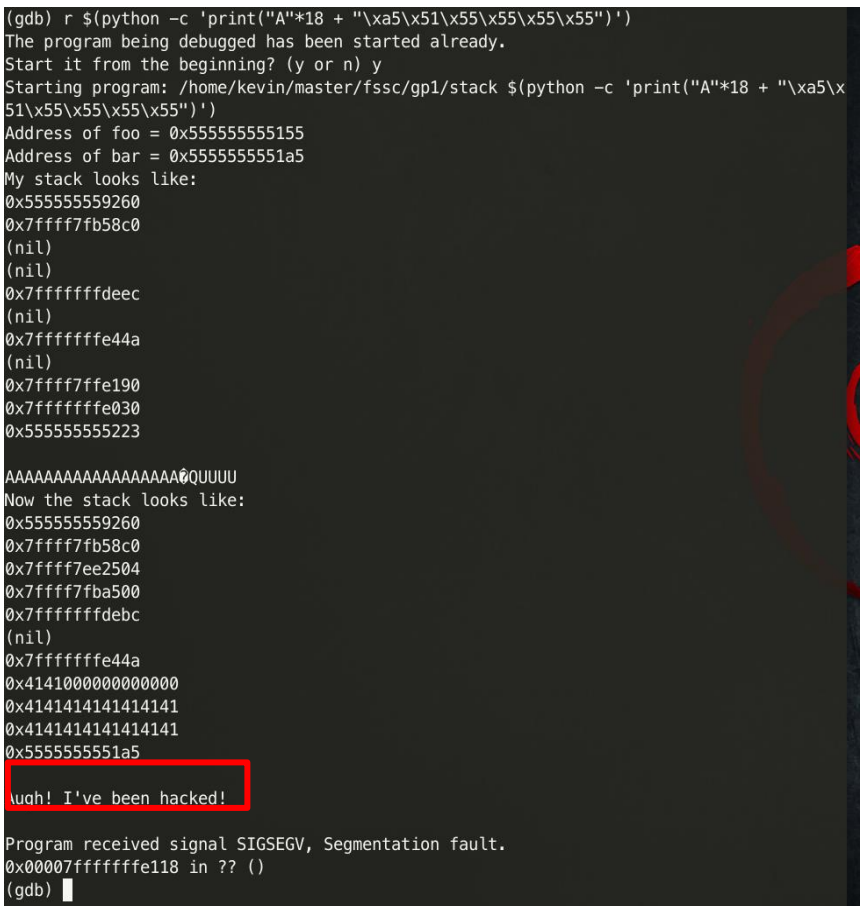
1. **Dado el siguiente código fuente en el lenguaje C, realice un análisis estático del mismo utilizando RATS y responda:**
    a. **¿Qué vulnerabilidades se detectan en este código?**

Lanzando el comado `$ rats --resultsonly -w 3 ej1.c rats` nos detecta dos vulnerabilidades de alto valor del mismo tipo, buffer overflow.

La primera vulnerabilidad (línea 7) a la hora de declarar la variable (`char buff[10]`) que es un array de caracteres de tamaño fijo. El contenido que se puede llegar a alojar en dicha variable puede ser superior a lo reservado en tiempo de compilación y producir un Stack Overflow.

La segunda vulnerabilidad (línea 9) se detecta a la hora de realizar un `strcpy`, esta función realiza una copia del contenido de una variable en otra variable, pero no comprueba el tamaño de ambas variables, si es indebidamente tratado es posible cambiar la ejecución del programa llamando a una zona de memoria que el usuario quiera, es decir nos avisa de un posible Buffer Overflow de tipo Stack Overflow.

Con conocimientos acerca de exploiting es posible modificar la dirección de retorno y llamar al método void bar(void) conociendo la dirección donde se localiza, se ha realizado como prueba de concepto, imprimiendo el contenido del método.



b. ¿Qué recomendaciones propone RATS para solucionarlas?

Para el primer riesgo es necesario asegurarnos de controlar el tamaño del contenido que se introduce en la variable `buff`.

En el caso de la segunda vulnerabilidad, el caso de la función `strcpy`, RATS recomienda asegurarse de que el segundo argumento que se le pasa a la función no copie más datos de los que puede albergar el primer argumento, en este caso 10.

c. Solucione las vulnerabilidades detectadas y vuelva a ejecutar el análisis. Presente una captura de pantalla con el resultado del análisis (antes y después de solucionar las vulnerabilidades)

Antes de solucionar las vulnerabilidades, RATS nos daba el siguiente análisis:

```
root@kali:/home/jorge/Escritorio# rats --resultsonly -w 3 ejemplo.c
ejemplo.c:7: High: fixed size local buffer
Extra care should be taken to ensure that character arrays that are allocated
on the stack are used safely.  They are prime targets for buffer overflow
attacks.

ejemplo.c:9: High: strcpy
Check to be sure that argument 2 passed to this function call will not copy
more data than can be handled, resulting in a buffer overflow.
```

Una vez se han corregido las vulnerabilidades aparece una vulnerabilidad de severidad baja para asegurarnos de que tiene suficiente espacio el buffer para copiar, en este caso hemos que se puedan copiar los 10 caracteres establecidos.

```
kevin@VirtualBox:~/fssc/gp1$ rats --resultsonly -w 3 ej1_mitigated.c
ej1_mitigated.c:11: Low: strlcpy
Double check that your buffer is as big as you specify
```

Para mitigar las vulnerabilidades encontradas por RATS hemos hecho que la variable `buff` no tenga un tamaño prefijado en tiempo de compilación y hemos cambiado la función `strcpy` por la función `strlcpy` porque limita la cantidad de caracteres a copiar.

```
void foo(constchar* input)
{
    char buf[];
    printf("My stack looks like:\n%p\n%p\n%p\n%p\n%p\n%p\n\n");
    strlcpy(buf, input, 10)
    printf("%s\n", buf);
    printf("Now the stack looks like:\n%p\n%p\n%p\n%p\n%p\n%p\n\n");
}
```

2. **Elija un lenguaje de programación soportado por RATS (Python, C/C++, PHP, Perl o Ruby), busque guías de buenas prácticas de programación y analice las principales recomendaciones para desarrollar código seguro el lenguaje escogido. Presente dos ejemplos de buenas prácticas de programación en el lenguaje escogido.**

Para el lenguaje de programación C dentro de las principales recomendaciones que dan las guías de buenas prácticas están la utilización de algunas funciones seguras en detrimento de otras funciones inseguras para prevenir de vulnerabilidades como condiciones de carrera, buffer overflows, format strings, etc. Además de la preferencia de utilización de algunas funciones por delante de otras, en las guías, se insta a los desarrolladores a controlar mucho las entradas de datos de los usuarios, tanto los tamaños como los tipos de datos.

Ejemplos:

- **Condiciones de carrera** al abrir ficheros:

En las guías, se recomienda comprobar que un fichero existe antes de empezar a trabajar con él. Con esta medida, se evitan las condiciones de carrera accediendo directamente al fichero en vez de sobrescribir el fichero.

- **Ataques de cadena de formato** (format string attacks) en la utilización de funciones como printf, fprintf, sprinf o snprintf:

Todas esas funciones esperan recibir el argumento de la cadena de formato. En las guías se recomienda que siempre se indique la cadena de formato cuando se utilicen todas estas funciones y que las entradas por teclado de los usuarios no se utilicen directamente como argumentos en estas funciones.

3. **Elija una vulnerabilidad en alguno de los lenguajes estudiados y realice un demostrador con la programación de la misma. Presente el resultado del análisis del código utilizando RATS, y el mismo código con la vulnerabilidad resuelta.**

Hemos elegido la vulnerabilidad de cadenas de formato (format string attack) para el lenguaje c. Para demostrar esta vulnerabilidad hemos creado un código vulnerable:

Vemos el resultado del análisis con RATS:

```
root@kali:/home/jorge/Escritorio# rats --resultsonly -w 3 formatstring.c
formatstring.c:6: High: printf
Check to be sure that the non-constant format string passed as argument 1 to
this function call does not come from an untrusted source that could have added
formatting characters that the code is not prepared to handle.
```

Vemos que detecta una vulnerabilidad de cadenas de formato en la función `printf`. Para mitigar la vulnerabilidad, utilizamos la cadena de formato en la función `printf`:

```c
#include <stdio.h>

int main(int argc, char **argv) {
    char *secreto = "Esto es un secreto!\n";

    printf("%s", argv[1]);

    return 0;
}
```

Con la mitigación realizada, volvemos a realizar el análisis del código con RATS:

```
root@kali:/home/jorge/Escritorio# rats -w 3 formatstring.c
Entries in perl database: 33
Entries in ruby database: 46
Entries in python database: 62
Entries in c database: 334
Entries in php database: 55
Analyzing formatstring.c
Total lines analyzed: 10
Total time 0.000238 seconds
42016 lines per second
```

Vemos que RATS ya no encuentra ninguna vulnerabilidad en el código después de haber realizado la mitigación.