

Overflows, underflows y otras fallas de seguridad

PROGRAMACIÓN SEGURA

Nuestro experto en seguridad nos comenta cuáles son las herramientas de software que podemos usar para auditar y securizar software. **POR KURT SEIFRIED**

Uno de los temas más recurrentes en el libro de Peter Seibel, *Coders at Work* [1] es: ¿Cómo ser un mejor programador? Esta idea me interesa sobremanera porque en el mundillo de la seguridad he podido comprobar que hay gente que comete sistemáticamente el mismo tipo de errores una y otra vez.

Después de cierto tiempo, cabría pensar que las personas aprenden a crear archivos temporales de forma segura. O quizá podrían al menos aprender a crear, acceder y eliminar buffers de memoria de forma segura; sin embargo, está claro que con muchos programadores (incluidos algunos muy buenos) no es el caso. Por tanto, la respuesta más obvia pasa por observar a las personas que sí escriben código seguro, emularlos y aprender de ellos. Pero, ¿cómo podemos hacerlo, especialmente cuando se dispone de poco tiempo y aún menos presupuesto?

A muchos programadores les encanta compartir. La belleza del conocimiento y la información radica en que una persona puede enseñar a otra, eternizando sus propios conocimientos. Por suerte, algunos programadores se han tomado la molestia de sintetizar sus conocimientos y su sabiduría a través de la creación de paquetes de software; de hecho, el

número de librerías disponibles a día de hoy para el programador sigue creciendo (gracias a estas personas, la mayoría de nosotros nunca tendrá que escribir un cliente HTTP o un parser de HTML). Entonces, ¿qué programas podemos utilizar para crear software más seguro y mejor?

Valgrind

Valgrind [2] probablemente sea el juego de herramientas de código abierto (licenciadas bajo GPL) más maduro para el análisis de código fuente y de binarios, no sólo orientado a problemas de seguridad, sino también a problemas de rendimiento (suelen ir ambos de la mano). Probablemente, el componente más útil a la hora de diagnosticar y solucionar problemas de seguridad sea *memcheck*.

El módulo *memcheck* encuentra toda la memoria que es accesible pero que no debería serlo, valores no inicializados utilizados de forma peligrosa, agujeros de memoria (siempre un potencial problema de denegación de servicio), y la liberación inadecuada de bloques de memoria (liberaciones dobles, liberaciones erróneas, etc.), detectando además cuándo nuestro programa pasa bloques de memoria solapados, de origen y de destino.

Instalación y Uso de Valgrind

La instalación de Valgrind en sistemas basados en RPM o en dpkg es trivial:

```
yum install valgrind
```

o

```
apt-get install valgrind
```

Compilar Valgrind desde los fuentes es igualmente sencillo:

```
cd valgrind
./autogen.sh
./configure --prefix=/ruta
make
make install
```

Uso de Valgrind

Aquí es donde las cosas se complican un poco. Valgrind genera una salida muy extensa. Parte de ella proviene de falsos positivos o cosas no demasiado peligrosas, y separar el polvo de la paja puede llevarnos cierto tiempo. Esto nos conduce directamente al Debian Bug Report Log 363516 [3] y a mi artículo sobre la falla de seguridad que se introdujo con esta modificación del código fuente [4]. A grandes rasgos, Valgrind advertía sobre el uso de memoria sin inicializar; en vez de usar un archivo de ocultación (para hacer que Valgrind pare de quejarse) o permanecer sentado hasta comprender realmente lo que hacía el código, el desarrollador optó por comentar sin más las líneas afectadas. Por desgracia, dicha acción tuvo como resultado un fondo de entropía predecible que en última instancia derivó en claves generadas por OpenSSL en Debian completamente predecibles y fáciles de averiguar

debido al bajo número de posibilidades existentes. Puesto que ese mismo es el caso de muchas herramientas potentes, usarlas sin un conocimiento completo puede hacer que las consecuencias lleguen a ser desastrosas. Hay un par de vídeos excelentes de SecurityTube [5] que cubren algunos de los aspectos básicos sobre el uso de Valgrind en Linux.

A aquellos que tengan que programar pero no confíen al cien por cien en que sus métodos de programación sean seguros, o que no tengan la experiencia suficiente como para usar Valgrind a fin de determinar verdaderamente que sus programas lo son, quizá les convenga simplemente usar un lenguaje de programación más seguro.

Usar un Lenguaje Más Seguro

Hay una solución relativamente simple que soluciona la mayoría de los problemas relativos a gestión de memoria, tales como ubicar memoria adecuadamente (uso de memoria sin inicializar, etc.), usar memoria de forma segura (desbordamientos de búfer, etc.) o garantizar que la memoria es destruida correctamente (liberaciones dobles, agujeros de memoria, etc.): Usar un lenguaje de programación con gestión de memoria integrada (como Python, Java, Perl, etc.). De este modo, en gran parte se evitan problemas como tener que preocuparse de la longitud de una cadena o de un array en el momento de crearlo. Simplemente se crea la cadena o el array y se le pasan los datos. El búfer se expande bajo demanda, y a la hora de leerlo, no se puede pasar de su final, ya que el intérprete del programa simplemente devuelve un error diciendo que no hay más datos o elementos que leer (en vez de leer áreas de memoria aleatorias sin quejarse). La desventaja de estos lenguajes es que ciertos problemas y programas no se prestan a ellos (casi todas las implementaciones de estos lenguajes están escritas en C, que es una pesadilla en lo que a gestión de memoria se refiere).

Auditando el Código Fuente: PyChecker

Además, como buenos usuarios de Unix de cinturón y tirantes, podemos usar una simple pero efectiva herramienta de comprobación de código fuente Python llamada *PyChecker*.

PyChecker [6] es una herramienta que sirve para auditar código fuente escrito en Python. Aunque el mismo intérprete de Python descubrirá la mayor parte de los errores de programación (lanzando excepciones, imprimiendo el error y saliendo), siempre es posible que se le escape algo. El uso de una variable global dentro de una clase, a diferencia de una variable propia (que sólo existe dentro de esa instancia de la clase y es, por tanto, mucho más segura en un entorno multihilo), puede llevar a todo tipo de problemas y condiciones de carrera difíciles de tratar. Sin embargo, *PyChecker* los detectará y nos informará sobre ellos inmediatamente. La instalación de *PyChecker* es sencilla:

```
easy_install PyChecker
```

Si esto no funcionase, siempre se puede instalar manualmente descargando el tarball de *PyChecker*, descomprimiéndolo y ejecutando el instalador a mano:

```
wget http://example.com/PyChecker-0.8.18.tar.gz
tar -xzf PyChecker-0.8.18.tar.gz
cd PyChecker-0.8.18
python setup.py install
```

Aprender a Programar de Forma Segura

Aún con todo, no hemos explicado nada desde el punto de vista educacional. Para programar de forma segura hay que entender, o al menos disponer de ciertos conocimientos al respecto, de los distintos tipos de fallas que hacen que un software sea inseguro. Dichas fallas van desde las más simples, como los distintos tipos de desbordamientos de búfer, hasta el esotérico “Uso de Función No-reentrant en un Contexto No Sincronizado”. El proyecto *CWE (Common Weakness Enumeration)* [7] está diseñado justo con ese fin; o sea, aportar una taxonomía completa de fallas de seguridad con descripciones, información acerca de soluciones y ejemplos de la falla. Algunas de las entradas de la *CWE* tienen código de ejemplo, pero por desgracia, la mayor parte de las entradas carecen de información sólida sobre cómo solucionar o cómo evitar el problema. Pero como dice G.I. Joe, “Ahora ya sabes, y el saber es la mitad de la batalla”.

El siguiente paso consiste en aprender las mecánicas y la mentalidad de la programación segura. Aunque existen literalmente decenas de libros disponibles, algunos de los cuales son bastante buenos, personalmente sigo prefiriendo los recursos en línea. Uno de los mejores documentos es *Secure Programming for Linux and Unix HOWTO* [8], de David A. Wheeler.

Otro proyecto que trata de mejorar la seguridad del software es *OWASP (Open Web Application Security Project)* [9]. Aunque está orientado sobre todo a aplicaciones web, la mayor parte de los contenidos de las guías *Development Guide*, *Code Review Guide* y *Testing Guide*, son aplicables al software no basado en web. Lo que es más importante aún, incluyen herramientas reales y documentación específica sobre cómo programar de forma segura, además de la “*WebGoat*”, una aplicación deliberadamente insegura que nos permite aprender de los errores de los demás.

Conclusión

No es tan difícil programar de forma segura; sobre todo se requiere disciplina. Esto implica no tomar atajos e invertir el tiempo necesario para comprender los efectos que los cambios tienen en el código (especialmente cuando el código original lo ha escrito otra persona); y lo que es más importante, comprender cómo interactúa nuestro código con otros códigos y sistemas. ■

RECURSOS

- [1] *Coders at Work*, de Peter Seibel, Apress, 2009, <http://www.codersatwork.com/>
- [2] Valgrind: <http://valgrind.org/>
- [3] Debian Bug report logs - #363516: <http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=363516>
- [4] *Caja Negra*, por Kurt Seifried, Linux Magazine, Número 44, pág. 08: http://www.linux-magazine.es/issue/44/008-009_Inseguridades44.pdf
- [5] SecurityTube: <http://www.securitytube.net/>
- [6] *PyChecker*: <http://pychecker.sourceforge.net/>
- [7] *CWE (Common Weakness Enumeration)*: <http://cwe.mitre.org/data/slices/2000.html>
- [8] *Secure Programming for Linux and Unix HOWTO*: <http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO.html>
- [9] OWASP: <http://www.owasp.org/>