Puntos de evolución

Una vez finalizada la idea inicial con la que se comienza un proyecto, siempre surgen ideas relacionadas con la labor llevada a cabo, bien sea para añadir, modificar o eliminar algún punto concreto. No iba a ser menos en este proyecto, en el que se han abarcado diversos campos.

Mejora del filtrado

En primer lugar, como mejoras de sigMatch se proponen dos caminos diversos pero siempre relacionados con la tasa de candidatos. Se recuerda al lector que esta tasa estaba íntimamente ligada al factor de mejora obtenido al integrar sigMatch, obteniendo una mejora mayor cuanto más pequeña era dicha tasa.

En principio, esta tasa podía ser alta –y por tanto producir un pobre rendimiento– por dos factores distintos. El primero de ellos se debía a que el número de falsos positivos obtenidos era también muy alto, lo que significaría que la mayoría de los paquetes candidatos no resultarían ser detectados como amenazas por la unidad de verificación definitiva.

Para corregir esto se propone implementar el método *data conscious* de sigMatch, que no ha sido integrado en este desarrollo. Este método propone un estudio previo del tráfico de la red y de las firmas, que clasifique los resúmenes que provocan más falsos positivos, para así descartarlos en la creación de una segunda estructura de búsqueda sigTree. De esta forma, esta estructura devolverá menos falsos positivos, disminuyendo así la tasa de candidatos.

Esta solución resulta muy buena cuando la distribución de bytes recibidos en una red suele permanecer constante, es decir el tipo tráfico suele mantenerse invariable. El problema resulta cuando este tipo de tráfico varía con el tiempo, en función del día o de las horas. En estos casos, se propone un desarrollo que controle el valor de la tasa de candidatos, para que cuando supere un cierto umbral se desactive el filtrado y se ejecute Snort de forma normal. De esta forma se evitarían los casos de pérdida de rendimiento.

El segundo factor que provoca una tasa de candidatos alta se debía a que la propia tasa de ocurrencias de la unidad de verificación final era muy alta, por lo que la única solución que se podía tomar sería deshabilitar el filtro temporalmente, hasta que la tasa de candidatos se restableciera.

Nuevas tecnologías

Ya se ha visto que las librerías IPP de Intel proporcionan funciones de búsqueda de patrones que aprovechan al máximo el potencial de sus procesadores. No obstante, estas funciones son para patrones simples.

Se sabe también, que en Snort, que las ocurrencias producidas en el algoritmo Aho-Corasick van a

parar a una función –bautizada como *Match*– que verifica si se cumple la regla asociada a dicho patrón.

Una regla puede contener un patrón de más de 20 caracteres, por lo que al entrar en Aho-Corasick seguramente haya sido truncado a 20 caracteres –como se especifica en el archivo de configuración de Snort–. Esto provoca que al encontrarse dicho patrón truncado, se deba comprobar que realmente se trata del patrón completo. Este es uno de los casos por los que la función *Match* se debe comprobar nuevamente un patrón cuando se produce una coincidencia en Aho-Corasick.

Por tanto, la función *Match*, entre otras tareas, realiza búsquedas simples de patrón. En este punto es donde se podrían implementar las funciones de búsqueda de patrones de Intel. El rendimiento final no sería muy alto para entornos que provoquen con tasas de ocurrencia bajas, pero en el caso de entornos agresivos, donde las ocurrencias pueden alcanzar de un 80% a un 90%, se podría mejorar significativamente el rendimiento, sin riesgo a obtener un peor rendimiento.

Más allá del filtrado

Conocer en profundidad el motor de búsqueda de patrones de Snort permite razonar de forma diversa ante una problemática definida.

Ya se ha explicado que cuando se produce una ocurrencia en Aho-Corasick, ésta se envía a la función *Match*, que se encarga de verificar la alerta. Al integrar sigMatch en Snort, el procedimiento es similar: sigMatch obtiene un candidato y lo envía hacia Aho-Corasick para que compruebe si verdaderamente se trata de una ocurrencia, en cuyo caso se enviaría a la función *Match* para comprobar si se trata de una alerta.

En un momento se estudió la idea de eliminar Aho-Corasick en ese paso intermedio, actuando sigMatch como un algoritmo de búsqueda multi-patrón más que se limita a enviar un candidato a la función *Match*. Esto ahorraría un paso intermedio, pero a su vez se incrementaría la carga de la última función implicada, que no se caracteriza por contar con un gran rendimiento.

También, se sabe que sigMatch no siempre es más eficiente que Aho-Corasick. De hecho, sólo lo es en entornos con una tasa baja de candidatos. Por lo que en entornos desfavorable el rendimiento se vería muy perjudicado, incluso más que combinando sigMatch y Aho-Corasick.

Estos dos motivos provocaron que se descartara la idea de quitar Aho-Corasick de la ecuación y dejar solo a sigMatch. Pero otras puertas se abrieron.

Es cierto que el rendimiento de sigMatch se ve muy perjudicado debido a su NFA inicial seguido de filtros Bloom y listas enlazadas de patrones cortos. Pero también es cierto que en condiciones favorables se convierte en un algoritmo muy rápido. Del estudio de la algoritmia realizado en el capítulo dos, así como de todo el conocimiento adquirido en Snort, se puede concluir que el cuello de botella en la estructura sigMatch es el filtro Bloom y la lista enlazada –esta conclusión también se puede sacar del capítulo siete, en el apartado de previsión, concretamente el punto dedicado al coste computacional–.

En definitiva, se tiene la estructura sigTree, cuya primera parte correspondiente al NFA es bastante rápida, mientras que la segunda parte tiene un rendimiento que deja que desear. Por otro lado, se sabe que la estructura DFA de Aho-Corasick tiene un rendimiento muy constante. Con estos datos, el resultado del problema no fue otro que proponer una nueva estructura de búsqueda basada en un

NFA inicial de tamaño reducido, seguido de un DFA, eliminando el filtro Bloom y la lista enlazada. Esta estructura prometería una margen de variación del rendimiento bastante menor que el de sigMatch, por lo que podría usarse en solitario como un algoritmo de búsqueda más.

Curiosamente, durante los días en los que se desarrollaba esta idea, se encontró un estudio elaborado por Chenqian Jiang, de la Universidad del Sur de California, titulado *Head-Body Partitioned String Matching for Deep Packet Inspection with Scalable and Attack-Resilient Performance*, en el cual se plasmaba el desarrollo de esta misma idea, mostrando unos resultados muy favorables.