

So how does nCircle detect legacy session renegotiation on a server? Well, essentially we mimic what any SSL client would attempt to do to renegotiate a TLS session. nCircle's detection obviously differs for TLSv1 vs. SSLv3 since they are different protocols, but the general flow is the same for both. For this post I'll be describing TLSv1 Session Renegotiation detection.

The most complex part about this detection is first establishing an encrypted tunnel, and if you don't care about how we do that, feel free to jump down to the last three paragraphs where I explain what happens after we establish the tunnel. To establish an encrypted tunnel, we first start by creating a pre-master secret, used for generating symmetric keys by both the client and the server, and storing it for later use. The pre-master secret consists of 46 random bytes and 2 bytes to identify the protocol version. The next step is then to craft a Client Hello packet, which will be used to initiate the TLS handshake process. The payload of the Client Hello packet includes information such as the packet length, a random seed, session ID length, session ID, compression method length, compression methods, and a complete list of all ciphers supported by the client. In our case, we don't need a session ID or a compression method, but we do add a variety of RSA cipher suites to our list of supported ciphers. After crafting the packet we send it off to the server, obviously in the clear, and await a response.

If the server accepts the conditions of our handshake and chooses to continue with the process, it will send us back a Server Hello, the server's certificate, and a Server Hello Done. If we don't receive a Server Hello at this point, then the TLS handshake has failed, an encrypted tunnel will not be established. Assuming we did get the Server Hello response, we parse the packet to retrieve the server's random seed, the session ID, and the cipher suite selected by the server from the list of cipher suites we sent it in the Client Hello. While parsing the accepted cipher suite we must store the cipher's key size, hash algorithm and encryption algorithm for later use. The key exchange algorithm will be RSA. We then parse the server's public certificate which we store as an X509 chain (a list of X509 certificates). At this point we must generate a symmetric key (our master secret). To do this we need the pre-master secret we created earlier, as well as the random seed generated by us (the client) earlier, and the random seed sent to us by the server. We supply these to a pseudo-random function, where the client's random seed and the server's random seed are concatenated together along with the label "master secret" to be used as the seed with the pre-master secret to create both an MD5 P hash as well as a SHA P hash. These two P hashes are then bitwise XOR'd together to give us our pseudo-random master secret. A P hash is created by expanding an HMAC (Hash-based Message Authentication Code) hash to an arbitrary length. We can now use this master secret to generate a key block. To create the key block, we supply our master secret to the pseudo-random function using the server random seed and client random seed concatenated together with the label "key expansion" as our seed. The key block length will be the twice the cipher key size, plus twice the hash length, plus twice the initialization vector length. Once the key block is created, we

slice it to obtain the client write MAC secret, server write MAC secret, client write key, server write key, client write initialization vector, and the server write initialization vector.

The next step in the process is to initiate a client key exchange, and to do this we must craft another packet. During this stage of the process we're going to send the server our pre-master secret which we generated earlier. To do this, we transform our pre-master secret into cipher text by encrypting it using the RSA algorithm, with the server's public key. We then construct the packet using this cipher text, and send it to the server. We must now generate both a change cipher spec message and a finished message. The change cipher spec message is simple, and is used to notify the server that any further communications after this will be encrypted under the CipherSpec and keys we have just negotiated. The finished message is a bit more complicated, and requires us to take the handshake data we've been storing (handshake data we've been sending the server as well as handshake data received from the server) throughout this handshaking process, and generate both an MD5 digest as well as a SHA digest from it and we will use both of them as our seed. The master secret is sliced in two, where we then create P hashes of each half using the seed from earlier, and we create a bitwise XOR of them. The first part to our finished message is now complete. A MAC, or Message Authentication Code, must now be calculated. To calculate a MAC, we first create a message containing the sequence number (0), the content type (handshake), protocol type (TLSv1), length of the finished message, and the finished message. We can now calculate our MAC by creating an HMAC hash using our client MAC secret stored earlier and the message we just created. Once we have calculated our MAC, we add it to our finished message and symmetrically encrypt the entire finished message using the session's master key. We then craft our packet using this encrypted finished message and finally send the packet off to the server, where we will then wait for a response.

We now expect two messages from the server, a change cipher spec message to indicate to us that any further communication from the server will be encrypted, and a finished message. When parsing the finished message, we can read the record layer in clear text, but the handshake layer will be encrypted. After trimming out the record layer from the packet, we symmetrically encrypt the remaining encrypted handshake layer (which decrypts it), and store it for later. We now calculate a MAC using the finished message we just parsed out and decrypted. We can use this newly calculated MAC and compare it against the MAC parsed from the decrypted record layer to verify that there isn't a server MAC error. We then generate another finished message using the session's master secret and compare it to the finished message parsed from the decrypted record layer to ensure that a server hash error hasn't occurred. We then update our handshake data with the finished message, and we now finally established a fully encrypted TLS tunnel.

Now that we have an encrypted tunnel, it's time to determine whether or not legacy session renegotiation is disabled on the target server. To do this we will craft another Client Hello packet, but this time we will symmetrically encrypt it since it will be sent over the encrypted tunnel instead of in the clear like the first Client Hello. Additionally the record layer must indicate that this SSL packet is an SSL Record Handshake, instead of SSL Application Data, otherwise the server wouldn't know to treat the packet as a renegotiation request. In an attack scenario, an attacker would get this Client Hello packet from an unknowing victim client who is merely attempting to establish a TLS session of their own. We send the encrypted packet, and wait for a server response. Once we receive the server response, we decrypt it and check to determine what kind of packet it is. If it's a Server Hello packet, we know that legacy session renegotiation is not disabled on the target server, since the server is willing to renegotiate. If this packet were of any other type, such as an alert message, then we know that the server has rejected the request to renegotiate and so legacy session renegotiation has been disabled on the target system. Once we make this determination, we send the server an alert close notify message to notify the server that we would like to close the session.

If you've followed along until this point, you may have noticed that we don't step through the entire renegotiation process, we instead only check if the server is willing to renegotiate. The reason for this is simple. Under no circumstance should legacy renegotiation be left enabled on the server, unless some legacy system or application requires it, in which case you must make a conscious decision to accept that risk. We've had customers in the past come to us claiming a false positive on this vulnerability, only for them later to realize that even though OpenSSL `s_client` may not have been able to renegotiate a session with their server, they hadn't actually disabled legacy renegotiation. If I were the systems administrator of a TLS enabled web server, I may have done my due diligence by running OpenSSL `s_client` to attempt to renegotiate a session, and if the renegotiation failed for whatever reason I may have incorrectly assumed that I disabled renegotiation on my server. If my security products didn't inform me that I was mistaken, I would feel that they weren't giving me all the information I needed to secure my systems. I would always take a defense-in-depth approach to security, and would want to be certain that my servers were configured such that under no circumstances would they ever be willing to renegotiate a TLS session using legacy renegotiation.

For our customers, I hope this gives you some insight into the steps nCircle takes to help ensure that this "feature" of TLS won't ever be abused in such a way so as to compromise the data of your users. For others, I hope this was at the very least semi-interesting, and perhaps even useful to you in helping you to write or improve your own detection. Again, nCircle VERT is always looking for good feedback on how to improve our detection so that we can best help keep our customers safeguarded from threats, and feedback on our Legacy TLS Session Renegotiation detection is certainly no exception.