MODULE *SPA_Attack_New*

This is the specification for the improved SDP architecture and algorithm which fixed the flaws related to service hidden feature. The defect study is aimed on the following materials: https://cloudsecurityalliance.org/artifacts/software-defined- perimeter-zero-trust-specification-v2/ http://www.cipherdyne.org/fwknop/

Author: *Dong.luming@zte.com.cn*

EXTENDS *FiniteSets*, *Sequences*, *Naturals*, *Integers*, *TLC*, *Bitwise*

The end point user's (*SDP* client) configuration, includes local *IP* and account *Info*.

CONSTANT *ClientCfg* @type: [*LoginID* $\mapsto$ String, *Key* $\mapsto$ Integer, *SrcIp* $\mapsto$ Integer];

The *SDP* controller's exposure service info, inludes listening *IP* and port.

CONSTANT *SDPSvrCfg* @type: [*IP* $\mapsto$ Integer, *Port* $\mapsto$ Integer];

The target server's exposure service info, inludes server *IP* and listening port.

CONSTANT *SvrCfg* @type: [*IP* $\mapsto$ Integer, *Port* $\mapsto$ Integer];

The attacker's configuration, inludes local *IP*.

CONSTANT *AttackerCfg* @type: [*SrcIp* $\mapsto$ Integer];

The match any type value for a *ACL Rule*.

CONSTANT *MATCH_ANY* @type: Integer;

For an user's socket link , the random local port range.

CONSTANT *USER_BASEPORT* @type: Integer;

For an attacker's socket link , the random local port range.

CONSTANT *ATTACKER_BASEPORT* @type: Integer;

If the attacker and user are in the same *LAN* with a shared public *IP* for *NAT*.

CONSTANT *NAT_FLAG* @type: *BOOL*;

The invalid authentication session *ID* value.

If a data access link with an invalid authentication session *ID*, it means we don't know the data access is resulted by which *Auth* ses

CONSTANT *UNKNOWN_AUTH_ID* @type: Integer;

If the legal user and attacker are in the same *LAN* with shared public *IP*, then the local port range after *SNAT* must not conflict w

ASSUME $(NAT\_FLAG = \text{TRUE} \Rightarrow AttackerCfg.SrcIp = ClientCfg.SrcIp \land USER\_BASEPORT \neq$

ASSUME $(SDPSvrCfg.IP \neq ClientCfg.SrcIp \land SDPSvrCfg.IP \neq AttackerCfg.SrcIp)$

ASSUME $(SvrCfg.IP \neq ClientCfg.SrcIp \land SvrCfg.IP \neq AttackerCfg.SrcIp)$

ASSUME $(SvrCfg.IP \neq SDPSvrCfg.IP)$

The variables related to legal user's state machine

The legal user's status indicates which session it is undergoing now.

VARIABLE $uState$ @type: {"Start_Auth", "Auth_End", "Auth_End_Connecting", "Connected"};

The legal user's *IP* address get from configuration.

VARIABLE uIP @type: Integer;

The legal user's *ID* for authentication.

VARIABLE uID @type: String;

The legal user's Secret *Key* for authentication.

VARIABLE Key @type: Integer;

The legal user's *Sync* counter value (Time Stamp) for *SDP* authentication, the counter increases randomly each auth s

VARIABLE $uTstamp$ @type: Integer;

The legal user's knowledge for *SDP* controller's info got from configuration.

VARIABLE $uSDPSvrInfo$ @type: $[IP \mapsto$ Integer, $Port \mapsto$ Integer];

The legal user's knowledge for target server's info got from configuration.

VARIABLE $uSvrInfo$ @type: $[IP \mapsto$ Integer, $Port \mapsto$ Integer];

The legal user's *TCP* links connected with target server for access.

VARIABLE $uTCPLinkSet$ @type: $Set([sIP \mapsto Integer,$
$sPort \mapsto$ Integer,
$dIP \mapsto$ Integer,
$dPort \mapsto$ Integer,
$State \mapsto$ {"SYN_SENT", "ESTABLISHED"},
$Retrans \mapsto$ {TRUE, FALSE}]);

The legal user's *Authenticaiton* sessions in history recorded in *Log.Each* session identified by a SPA message.

VARIABLE $uAuthSession$ @type: $Set([MsgID \mapsto$ "SPA_AUTH",

$sIP \mapsto uIP$,
$sPort \mapsto RandomPort(uTstamp, USER\_BASEPORT)$,
$dIP \mapsto uSDPSvrInfo.IP$, * The SDP Controller's IP and port for SPA protocol
$dPort \mapsto uSDPSvrInfo.Port$,
$ClientID \mapsto uID$, $Tstamp \mapsto uTstamp$, * increased each session to anti Replay
$CliIP \mapsto Encrypt(uIP, Key)$, * < $CliIP$,$CliPort$,$SvrIP$,$SvrPort$ > is the connecting TCP link info to the target server
$CliPort \mapsto Encrypt(RandomPort(uTstamp, USER\_BASEPORT) + 1, Key)$, * the random local port of TCP connection for data access
$SvrIP \mapsto Encrypt(uSvrInfo.IP, Key)$, * Target Server's exposure service $Info$, need to kept secret
$SvrPort \mapsto Encrypt(uSvrInfo.Port, Key)$,
$HMAC \mapsto CalcHMAC(uIP, uID, uTstamp, Encrypt(uSvrInfo.IP, Key), Encrypt(uSvrInfo.Port, Key), Ke$
, * $HMAC$ of payload
$Type \mapsto Set($"User", "Attacker"$)$]); * Flag to indicate this message is built by legal user or attacker * this flag not invloved in inter-operation between SDP protocol entities,only for statistic

The legal user equipment's packets channnel for recieving data plane packets, corresponds to its physical NIC.

VARIABLE $uChannel$ @type: Sequence of TCP Packets $Seq([sIP \mapsto p.dIP$, * TCP *packets for data access, for this model*
$sPort \mapsto p.dPort$, * simulate the data plane access stream only by TCP connection proceudre
$dIP \mapsto p.sIP$, * IE. if user establish a TCP connection with target server, that
$dPort \mapsto p.sPort$, *means a sucessful data access session.
$Flg \mapsto Set($"TCP_SYN", "TCP_SYN_ACK", "TCP_ACK"$)$,
* TCP handshake packets type.
$Type \mapsto Set($"User", "Attacker"$)$]; * Flag to indicate this access is initiated by legal user or attacker * this flag not invloved in inter-operation between SDP protocol entities,only for statistic

The legal $User$'s private variables ($uChannel$ is public variable of user, for other entity can operate and modify $uChannel$ variable o

$$user\_vars \triangleq \langle uState, uIP, uID, Key, uTstamp, uSDPSvrInfo, uSvrInfo, uTCPLinkSet, uAuthSession\rangle$$

The variables related to SDP Server's (SDP Controller) state machine

The SDP controller's status indicates this entity's service is available or faulty.

VARIABLE $SDPSvrState$ @type: $Set($"Work"$)$

The SDP controller successfully processed $Auth$ sessions in history recorded in Log.

VARIABLE $SDPSucSession$ @type: $uAuthSession$

The Legal user accounts info recorded in SDP controller's IAM system.

VARIABLE $Account$ @type: $Set([ClientID \mapsto ClientCfg.LoginID,$
$Key \mapsto ClientCfg.Key])$

The SDP controller's exposure SPA service info .

VARIABLE $SDPSvrInfo$ @type: $[IP \mapsto SDPSvrCfg.IP,\ Port \mapsto SDPSvrCfg.Port]$

The number of replay attack messages inspected by SDP controller

VARIABLE $ReplayCount$ @type: Integer;

The number of spoof attack messages inspected by SDP controller

VARIABLE $SpoofCount$ @type: Integer;

The replay attack $Auth$ sessions inspected by SDP controller in history recorded in Log.

VARIABLE $ReplaySession$ @type: $uAuthSession$;

The spoof attack $Auth$ sessions inspected by SDP controller in history recorded in Log.

VARIABLE $SpoofSession$ @type: $uAuthSession$;

SDP controller's packets channnel for recieving control plane $Auth$ messages, corresponds to its physical NIC.

VARIABLE $AuthChannel$ @type: Sequence of SPA $Auth$ Packets $Seq([MsgID \mapsto$ "SPA_AUTH",
$sIP \mapsto uIP,$
$sPort \mapsto RandomPort(uTstamp,\ USER\_BASEPORT),$
$dIP \mapsto uSDPSvrInfo.IP,$ * The SDP Controller's IP and port for SPA protocol
$dPort \mapsto uSDPSvrInfo.Port,$
$ClientID \mapsto uID,\ Tstamp \mapsto uTstamp,$ * increased each session to anti Replay
$CliIP \mapsto Encrypt(uIP,\ Key),$ * $< CliIP, CliPort, SvrIP, SvrPort >$ is the connecting TCP link info to the target server
$CliPort \mapsto Encrypt(RandomPort(uTstamp,\ USER\_BASEPORT) + 1,\ Key),$ * the random local port of TCP connection for data access
$SvrIP \mapsto Encrypt(uSvrInfo.IP,\ Key),$ * Target Server's exposure service $Info$, need to kept secret
$SvrPort \mapsto Encrypt(uSvrInfo.Port,\ Key),$
$HMAC \mapsto CalcHMAC(uIP,\ uID,\ uTstamp,\ Encrypt(uSvrInfo.IP,\ Key),\ Encrypt(uSvrInfo.P$
, * $HMAC$ of payload
$Type \mapsto Set($"User", "Attacker"$)]);$ * Flag to indicate this message is built by legal user or attacker * this flag not invloved in inter-operation between SDP protocol entities,only for statistic;

The SDP controller's private variables ($AuthChannel$ is public variable of SDP controller, for other entity can operat

$sdpsvr\_vars \triangleq \langle SDPSvrState,\ SDPSucSession,\ Account,\ SDPSvrInfo,\ ReplayCount,\ SpoofCount,$

The variables related to FireWall's state machine

The *FireWall*'s status indicates this entity's service is available or faulty.

The *FireWall* works in deny mode by default.

VARIABLE *FwState* @type: *Set*("Work")

Current *Acl Rule* Set maintained by the *FireWall* for data plane traffic.

VARIABLE *AclRuleSet* @type: *Set*([*sIP* ↦ *Integer*,
sPort ↦ Integer, \ * the value can be *MATCH_ANY*,
dIP ↦ Integer,
dPort ↦ Integer,
protocol ↦ "TCP",
action ↦ "Accept"])

The aged *Acl* Rules in history recorded in *FireWall*'s log.

VARIABLE *AgedRuleSet* @type: *Set*([*sIP* ↦ *Integer*,
sPort ↦ Integer, \ * the value can be *MATCH_ANY*,
dIP ↦ Integer,
dPort ↦ Integer,
protocol ↦ "TCP",
action ↦ "Accept"])

The dropped packets by *FireWall* in history recorded in log.

VARIABLE *DropPackets* @type: *Set*([*sIP* ↦ *p.dIP*, \ * *Only data plane TCP packets are processed by FireWall*
sPort ↦ *p.dPort*,
dIP ↦ *p.sIP*,
dPort ↦ *p.sPort*,
Flg ↦ *Set*("TCP_SYN", "TCP_SYN_ACK", "TCP_ACK"), \ * *TCP* handshake packets type.
Type ↦ *Set*("User", "Attacker")];)

FireWall's control plane channel for receiving *Openflow* instruction from *SDP* controller to configure data access *Acl Rule*, corresp

VARIABLE *FwCtlChannel* @type: Sequence of *Acl* config instructions *Seq*([*Rule* ↦ *AclRule*, *op* ↦ *Set*("Add", "*Del*")])

FireWall's ingress data plane channnel for receiving packets from end point entities , corresponds to one of its physical *NIC*.

VARIABLE *FwDataChannel* @type: Sequence of Data Packets *Seq*([*sIP* ↦ *p.dIP*, \ * *Only data plane TCP packets are pr*
sPort ↦ *p.dPort*,
dIP ↦ *p.sIP*,
dPort ↦ *p.sPort*,
Flg ↦ *Set*("TCP_SYN", "TCP_SYN_ACK", "TCP_ACK"), \ * *TCP* handshake packets type.
Type ↦ *Set*("User", "Attacker")];)

The *FireWall*'s private variables (*FwDataChannel* and *FwCtlChannel* are public variable of *FW*, for other entity can operate and

$fw\_vars \triangleq \langle FwState, AclRuleSet, AgedRuleSet, DropPackets \rangle$

The variables related to Attacker's state machine

The *Attacker*'s status indicates this entity's is spying or not.

VARIABLE $aState$ @type: Set("Listen")

The *Attacker*'s current knowledge about legal user's auth action learned by sniffing legal user's auth message.

VARIABLE $AuthKnowledge$ @type: $uAuthSession$

The *Attacker* initiated SPA attack sesssions in history recorded in log. Each session is identified by a fake SPA messag

VARIABLE $aSession$ @type: $uAuthSession$

The *Attacker* initiated TCP connections towards the target server. Each link corresponds to an inspection attack to t

VARIABLE $aTCPLinkSet$ @type:Set([$sIP \mapsto Integer$,
$sPort \mapsto$ Integer,
$dIP \mapsto$ Integer,
$dPort \mapsto$ Integer,
$State \mapsto$ {"SYN_SENT", "ESTABLISHED"}
$AuthID \mapsto$ Integer] \ * The $AuthID$ is used for relating to a captured auth message
) \ * For this model, once the attacker spy a SPA message, it will undertake a data attack to the target server.
\ * The value $UNKNOWN\_AUTH\_ID$ indicates the attack is not originate from a captured auth message, but a captured data message

The number of sucessfully sniffed SPA messages by attacker

VARIABLE $sniffCount$ @type: Integer;

All the successfully sniffed SPA messages by attacker in history recorded in log

VARIABLE $CapAuthMsg$ @type: $uAuthSession$;

Attacker maintained increasing sequence number to build local port field for TCP links of different $Dtection$ attack.

VARIABLE $aCounter$ @type: Integer;

Attacker's IP address, which is got by configuration.

If $NAT\_FLAG =$ TRUE, then attacker and legal user located in the same LAN and share same public IP ($aIP = uIP$).

VARIABLE aIP @type: Integer;

The *Attacker*'s current knowledge about legal user's data access learned by sniffing legal user's TCP handshake packet

VARIABLE $DataKnowledge$ @type: Set([$sIP \mapsto p.dIP$, \ * *Only data plane TCP packets are processed by F*
$sPort \mapsto p.dPort$,

$dIP \mapsto p.sIP$,
$dPort \mapsto p.sPort$,
$Flg \mapsto Set($"TCP_SYN", "TCP_SYN_ACK", "TCP_ACK"$)$, * TCP handshake packets type.
$Type \mapsto Set($"User", "Attacker"$)])$

All the successfully sniffed user data packets by attacker in history recorded in log

VARIABLE $CapDataMsg$ @type: $DataKnowledge$

The attacker's packets channnel for recieving data plane packets, corresponds to its physical NIC.

VARIABLE $aChannel$ @type: $uChannel$

The attacker's private variables ($aChannel$ is public variable of attacker, for other entity can operate and modify $aChannel$ variable

$attacker\_vars \triangleq \langle aState, AuthKnowledge, aSession, aTCPLinkSet, sniffCount, CapAuthMsg, aCounter, aI$

The variables related to target service server's state machine

The target server's status indicates this entity's service is available or faulty.

VARIABLE $sState$ @type: $Set($"Listen"$)$

The TCP socket maintained in server side initiated from end points towards target server.

VARIABLE $sTCPLinkSet$ @type: $Set([sIP \mapsto p.dIP$, * *Only data plane TCP packets are processed by FireWall*
$sPort \mapsto p.dPort$,
$dIP \mapsto p.sIP$,
$dPort \mapsto p.sPort$,
$Flg \mapsto Set($"TCP_SYN", "TCP_SYN_ACK", "TCP_ACK"$)$, * TCP handshake packets type.
$Type \mapsto Set($"User", "Attacker"$)])$

The target server's exposure service info got from configuration.

VARIABLE $sSvrInfo$ @type: $[IP \mapsto SvrCfg.IP, Port \mapsto SvrCfg.Port]$

The server's packets channnel for recieving data plane packets from endpoint equipments, corresponds to its physical NIC.

VARIABLE $sChannel$ @type: $uChannel$

The target server's private variables ($sChannel$ is public variable of server, for other entity can operate and modify $sChannel$ variab

$server\_vars \triangleq \langle sState, sTCPLinkSet, sSvrInfo\rangle$

All the public variables of the model

$Public\_vars \triangleq \langle uChannel, AuthChannel, FwCtlChannel, FwDataChannel, aChannel, sChannel\rangle$

All the variables that consititute the global state machine

$vars \triangleq \langle user\_vars,\ sdpsvr\_vars,\ fw\_vars,\ attacker\_vars,\ server\_vars,\ Public\_vars\rangle$

Common functions and operators

Sequence to Set

$\text{RECURSIVE } Seq2Set(\_)$

$Seq2Set(S) \triangleq$

$\quad \text{IF } S = \langle\rangle \text{ THEN } \{\}$

$\quad\ \text{ELSE}$

$\qquad \text{LET } i \triangleq Head(S)$

$\qquad \text{IN } \quad \{i\} \cup Seq2Set(Tail(S))$

Select local port randomly when client create socket connection

$RandomPort(count,\ base) \triangleq (\text{CHOOSE } x \in (count + base) \,..\, (base + 100) : \text{TRUE})$

simulate Symmetric-key algorithm: *Encrypt* function, this operator simplified by *XOR* operation

$Encrypt(d,\ k) \triangleq$

$\quad \text{LET RECURSIVE } XorPureR(\_,\ \_,\ \_,\ \_)$

$\qquad XorPureR(x,\ y,\ n,\ m) \triangleq$

$\qquad\qquad \text{IF } m = 0$

$\qquad\qquad\ \text{THEN } 0$

$\qquad\qquad\ \text{ELSE LET } exp \triangleq 2^n$

$\qquad\qquad\qquad\quad \text{IN } \quad exp * (((x \div exp) + (y \div exp))\%2)$

$\qquad\qquad\qquad\qquad\quad + XorPureR(x,\ y,\ n+1,\ m \div 2)$

$\quad \text{IN } \quad \text{IF } d \geq k \text{ THEN } XorPureR(d,\ k,\ 0,\ d) \text{ ELSE } XorPureR(k,\ d,\ 0,\ k)$

simulate Symmetric-key algorithm: Decrypt function

$DeCrypt(d,\ k) \triangleq Encrypt(d,\ k)$

simulate *HMAC* function for improved SPA message

$CalcHMAC(n1,\ n2,\ n3,\ n4,\ n5,\ n6,\ n7,\ key) \triangleq Encrypt(n1 + n2 + n3 + n4 + n5 + n6 + n7,\ key$

Init state description of legal user

User *Init*: Read configuration and ready to launch an access to target server

the init state is ready to start a auth session.

$UsrInit \triangleq \ \wedge uState = \text{“Start\_Auth”}$

$\land\ uID = \ ClientCfg.LoginID$

$\land\ Key = ClientCfg.Key$

$\land\ uIP\ = ClientCfg.SrcIp$

$\land\ uTstamp = 0$

$\land\ uSDPSvrInfo = [IP \mapsto SDPSvrCfg.IP,\ Port \mapsto SDPSvrCfg.Port]$

$\land\ uSvrInfo = [IP \mapsto SvrCfg.IP,\ Port \mapsto SvrCfg.Port]$

$\land\ uTCPLinkSet = \{\}$

$\land\ uChannel = \langle\rangle$

$\land\ uAuthSession = \{\}$

Next state actions of legal user

$UsrBulidTcpSynPkt\ \triangleq$

$[sIP \mapsto uIP,$

$\ sPort \mapsto RandomPort(uTstamp,\ USER\_BASEPORT) + 1,$ the new data access corresponds to the latest auth

$\ dIP \mapsto uSvrInfo.IP,$

$\ dPort \mapsto uSvrInfo.Port,$

$\ Flg \mapsto$ "TCP_SYN",

$\ Type \mapsto$ "User"$]$

Action 1: *UsrConnectServerEnhance*

Legal user perform enhanced SPA auth which is triggered by and syncronized with the data plane *TCP* connecting event to target se

Variables changed: < *uState*,*uAuthSession*,*uTstamp*,*AuthChannel*,*uTCPLinkSet*,*FwDataChannel* >

$UsrConnectServerEnhance\ \triangleq$

$\land\ uState =$ "Start_Auth"

$\land\ uState' =$ "Auth_End_Connecting"

$\land\ uTstamp' = uTstamp + 1$ *uTstamp* increases each session for anti-replay.

$\land\ AuthChannel' = Append(AuthChannel,$

$[MsgID \mapsto$ "SPA_AUTH",

$\ sIP \mapsto uIP,$

$\ sPort \mapsto RandomPort(uTstamp,\ USER\_BASEPORT),$ the random port of S

$\ dIP \mapsto uSDPSvrInfo.IP,$

$\ dPort \mapsto uSDPSvrInfo.Port,$

$\ ClientID \mapsto uID,$

$Tstamp \mapsto uTstamp,$
$CliIP \mapsto Encrypt(uIP, Key),$ $< CliIP, CliPort, SvrIP, SvrPort >$ is t
$CliPort \mapsto Encrypt(RandomPort(uTstamp, USER\_BASEPORT)$
$SvrIP \mapsto Encrypt(uSvrInfo.IP, Key),$
$SvrPort \mapsto Encrypt(uSvrInfo.Port, Key),$
$HMAC \mapsto CalcHMAC(uIP, uID, uTstamp, Encrypt(uIP, Key)$
$Type \mapsto$ "User"]
)

$\wedge\ uAuthSession' = uAuthSession \cup \{Head(AuthChannel')\}$ Auth session is recorded in *Log*
$\wedge\ uTCPLinkSet = \{\}$
$\wedge\ uTCPLinkSet' = \{$ We assume the user only launch one data access session.
[$sIP \mapsto UsrBulidTcpSynPkt.sIP,$
$sPort \mapsto UsrBulidTcpSynPkt.sPort,$
$dIP \mapsto UsrBulidTcpSynPkt.dIP,$
$dPort \mapsto UsrBulidTcpSynPkt.dPort,$
$State \mapsto$ "SYN_SENT", Create new *TCP* socket corresponds to the latest *Auth* session, *TCP* lin
$Retrans \mapsto$ FALSE
] }
$\wedge\ FwDataChannel' = Append(FwDataChannel, UsrBulidTcpSynPkt)$ Send *TCP SYN* packet to *F*
$\wedge$ UNCHANGED $\langle uIP, uID, Key, uSDPSvrInfo, uSvrInfo\rangle$
$\wedge$ UNCHANGED $sdpsvr\_vars$
$\wedge$ UNCHANGED fw_vars
$\wedge$ UNCHANGED $attacker\_vars$
$\wedge$ UNCHANGED $server\_vars$
$\wedge$ UNCHANGED $\langle uChannel, FwCtlChannel, aChannel, sChannel\rangle$

Action 2: *UsrRcvSynAck*

Legal user receive *TCP SYN Ack* packet from target server which indicates data *TCP* link establised. This represents the user has successfully fufilled a data access.

Variables changed: $< uState, uTCPLinkSet, uChannel, FwDataChannel >$

$HasMatchLink(p, LinkSet) \triangleq$
$\exists\, x \in LinkSet : \ \wedge\ p.sIP = x.dIP$

$\wedge\ p.sPort = x.dPort$
$\wedge\ p.dIP = x.sIP$
$\wedge\ p.dPort = x.sPort$

$GetMatchLink(p,\ LinkSet) \triangleq$ get match TCB (TCP control Block) for a received TCP packet

$\text{CHOOSE } x \in LinkSet : \wedge\ p.sIP = x.dIP$
$\wedge\ p.sPort = x.dPort$
$\wedge\ p.dIP = x.sIP$
$\wedge\ p.dPort = x.sPort$

$EndPointBulidTcpAckPkt(p,\ t) \triangleq$ End point equipments might be a legal user or attacker

$[sIP \mapsto p.dIP,$
$sPort \mapsto p.dPort,$
$dIP \mapsto p.sIP,$
$dPort \mapsto p.sPort,$
$Flg \mapsto$ "TCP_ACK",
$Type \mapsto t]$

$UsrRcvSynAck \triangleq$

$\wedge\ (\vee\ uState =$ "Auth_End_Connecting"
$\vee\ uState =$ "Auth_End_Reconnecting"
$)$
$\wedge\ uTCPLinkSet \neq \{\}$
$\wedge\ uChannel \neq \langle\rangle$
$\wedge\ Head(uChannel).Flg =$ "TCP_SYN_ACK"
$\wedge\ Head(uChannel).Type =$ "User"
$\wedge\ HasMatchLink(Head(uChannel),\ uTCPLinkSet)$ Receive TCP_SYN_ACK from target server and match the conne
$\wedge\ uTCPLinkSet' = (uTCPLinkSet \setminus \{GetMatchLink(Head(uChannel),\ uTCPLinkSet)\})$
$\cup \{[sIP \mapsto GetMatchLink(Head(uChannel),\ uTCPLinkSet).sIP,$
$sPort \mapsto GetMatchLink(Head(uChannel),\ uTCPLinkSet).sPort,$
$dIP \mapsto GetMatchLink(Head(uChannel),\ uTCPLinkSet).dIP,$
$dPort \mapsto GetMatchLink(Head(uChannel),\ uTCPLinkSet).dPort,$
$State \mapsto$ "ESTABLISHED", Updata TCP link status to established
$Retrans \mapsto GetMatchLink(Head(uChannel),\ uTCPLinkSet).Retrans$

$]$

$\}$

$\wedge\ uState' = \text{"Connected"}$ The user successfully access the target server

$\wedge\ uChannel' = Tail(uChannel)$ Send *TCP ACK* packet (the last step of hand shake) to target server

$\wedge\ FwDataChannel' = Append(FwDataChannel,\ EndPointBulidTcpAckPkt(Head(uChannel),\ \text{"}$

$\wedge$ UNCHANGED $\langle uIP,\ uID,\ Key,\ uTstamp,\ uSDPSvrInfo,\ uSvrInfo,\ uAuthSession\rangle$

$\wedge$ UNCHANGED $sdpsvr\_vars$

$\wedge$ UNCHANGED fw_vars

$\wedge$ UNCHANGED $attacker\_vars$

$\wedge$ UNCHANGED $server\_vars$

$\wedge$ UNCHANGED $\langle AuthChannel,\ FwCtlChannel,\ aChannel,\ sChannel\rangle$

Action 3: *UsrReConnectServer*

If *TCP SYN* pakcets sent from legal user to target server dropped by the *FireWall* due to

the corresponding *ACL Rule* not configed yet, then legal user as *TCP* client will re-send *SYN* packet

to simulate the re-transmission mechanism of *TCP* protocol .

To simplify the model, we just trigger the retransmisson action only after the related *ACL Rule* is configed.

Variables changed: $< FwDataChannel, uState, uTCPLinkSet >$

Whether there exists 4 Tuple *ACL Rule* in *AclSet* that match the given *TCP* link l

$TcpLnkHasMatchAcl(l,\ AclSet) \triangleq$

IF $AclSet = \{\}$

THEN

FALSE

ELSE

$\exists\, r \in AclSet :$

$(\wedge\ r.sIP\ \ = l.sIP$

$\wedge\ r.sPort = l.sPort$

$\wedge\ r.dIP\ \ = l.dIP$

$\wedge\ r.dPort\ \ = l.dPort$

$\wedge\ r.action\ \ = \text{"Accept"}$

$)$

Whether the fire wall has packets dropping record for the given TCP link x.

$$
\begin{aligned}
& WithDropPkts(x) \triangleq \\
& \quad \exists\, p \in DropPackets : \land\, p.sIP = x.sIP \\
& \qquad\qquad\qquad\qquad\quad \land\, p.sPort = x.sPort \\
& \qquad\qquad\qquad\qquad\quad \land\, p.dIP = x.dIP \\
& \qquad\qquad\qquad\qquad\quad \land\, p.dPort = x.dPort
\end{aligned}
$$

$$SYN\_Timeout\_Lnk \triangleq \textsc{choose}\ x \in uTCPLinkSet : (x.State = \text{"SYN\_SENT"} \land WithDropPkts(x))$$

$$
\begin{aligned}
& UsrReConnectServer \triangleq \\
& \quad \land\, uState = \text{"Auth\_End\_Connecting"} \\
& \quad \land\, uState' = \text{"Auth\_End\_Reconnecting"} \\
& \quad \land\, uTCPLinkSet \neq \{\} \\
& \quad \land\, \exists\, x \in uTCPLinkSet : (x.State = \text{"SYN\_SENT"} \land WithDropPkts(x)) \\
& \quad \land\, TcpLnkHasMatchAcl(SYN\_Timeout\_Lnk,\ AclRuleSet) \\
& \quad \land\, FwDataChannel' = Append(FwDataChannel, \\
& \qquad [sIP \mapsto SYN\_Timeout\_Lnk.sIP, \\
& \qquad\ sPort \mapsto SYN\_Timeout\_Lnk.sPort, \\
& \qquad\ dIP \mapsto SYN\_Timeout\_Lnk.dIP, \\
& \qquad\ dPort \mapsto SYN\_Timeout\_Lnk.dPort, \\
& \qquad\ Flg \mapsto \text{"TCP\_SYN"}, \\
& \qquad\ Type \mapsto \text{"User"}]\ )
\end{aligned}
$$

Resend TCP SYN packet to $FireWall$.

$$
\begin{aligned}
& \quad \land\, uTCPLinkSet' = (uTCPLinkSet \setminus \{SYN\_Timeout\_Lnk\}) \\
& \qquad\qquad\qquad\quad \cup \{[sIP \mapsto SYN\_Timeout\_Lnk.sIP, \\
& \qquad\qquad\qquad\qquad\ sPort \mapsto SYN\_Timeout\_Lnk.sPort, \\
& \qquad\qquad\qquad\qquad\ dIP \mapsto SYN\_Timeout\_Lnk.dIP, \\
& \qquad\qquad\qquad\qquad\ dPort \mapsto SYN\_Timeout\_Lnk.dPort, \\
& \qquad\qquad\qquad\qquad\ State \mapsto SYN\_Timeout\_Lnk.State, \\
& \qquad\qquad\qquad\qquad\ Retrans \mapsto \text{TRUE}
\end{aligned}
$$

to record the retansmission event ever happened in link setup process

$$
\begin{aligned}
& \qquad\qquad\qquad\qquad] \\
& \qquad\qquad\qquad\qquad \} \\
& \quad \land\, \textsc{unchanged}\ \langle uIP,\ uID,\ Key,\ uTstamp,\ uSDPSvrInfo,\ uSvrInfo,\ uAuthSession \rangle \\
& \quad \land\, \textsc{unchanged}\ sdpsvr\_vars \\
& \quad \land\, \textsc{unchanged}\ fw\_vars
\end{aligned}
$$

$\wedge$ UNCHANGED $attacker\_vars$

$\wedge$ UNCHANGED $server\_vars$

$\wedge$ UNCHANGED $\langle uChannel, AuthChannel, FwCtlChannel, aChannel, sChannel \rangle$

Init state description of SDP Controller

SDP Controller *Init*: Read configuration and ready to provide SPA auth service.

$SDPSvrInit \triangleq$ $\wedge$ $SDPSvrState =$ "Work"

$\wedge$ $SDPSucSession = \{\}$

$\wedge$ $Account = \{[ClientID \mapsto ClientCfg.LoginID, Key \mapsto ClientCfg.Key]\}$ Load user

$\wedge$ $SDPSvrInfo = [IP \mapsto SDPSvrCfg.IP, Port \mapsto SDPSvrCfg.Port]$ Service *IP* and p

$\wedge$ $AuthChannel = \langle\rangle$

$\wedge$ $ReplayCount = 0$

$\wedge$ $SpoofCount = 0$

$\wedge$ $ReplaySession = \{\}$

$\wedge$ $SpoofSession = \{\}$

Next state actions of SDP Controller

Action 3: *SDPSvrProcSpaAuthEx*

SDP Controller process received improved version SPA message.

Scenario 3: Request from legal user, controller then instruct firewall to admit data access after authenticaiton.

Scenario 1 2: controller recognize spoof and replay attack.

Variables changed: $< AuthChannel, SDPSucSession, ReplaySession, SpoofSession, ReplayCount, SpoofCount, FwCtlCh$

if a coming SPA message *SN* match the history message recorded in anti-replay window

then it must be recognized as a replay attack packet.

$FindAntiReplay(msg, wnd) \triangleq$

IF $\exists\, r \in wnd : (msg.ClientID = r.ClientID \wedge msg.Tstamp = r.Tstamp)$

THEN

TRUE

ELSE

FALSE

For a recognized replay attack message, *SDP* controller drop it and recorded in the log.

$SDPSvrAntiReplayAtk \triangleq$

$\land AuthChannel' = Tail(AuthChannel)$ Drop packet

$\land ReplayCount' = ReplayCount + 1$ Increase statistics

$\land ReplaySession' = ReplaySession \cup \{Head(AuthChannel)\}$ Update log

For a recognized spoof attack message, *SDP* controller drop it and recorded in the log.

$SDPSvrAntiSpoof \triangleq$

$\land AuthChannel' = Tail(AuthChannel)$ Drop packet

$\land SpoofCount' = SpoofCount + 1$ Increase statistics

$\land SpoofSession' = SpoofSession \cup \{Head(AuthChannel)\}$ Update log

SDP controller implement authenticaiton triggered by a received SPA message

The authentication is implemented by recaculate the *HMAC* according the user account *Info*

$SpaProcAuth(msg, accounts) \triangleq$

$\exists a \in accounts : (\land a.ClientID = msg.ClientID$ user *ID* must match

Recaclulate the *HMAC* value by using local stored user *Key* and then compare the value of corresponding field in SPA packet.

$\land CalcHMAC(msg.sIP, msg.ClientID, msg.Tstamp, msg.CliIP, msg.CliPort, msg.Svr1$

$)$

Get the correspond key by user *ID* from *IAM* stored accounts

$GetKey(id, accounts) \triangleq (\text{CHOOSE } a \in accounts : a.ClientID = id).Key$

SDP controller instruct *FireWall* to config *Acl Rule* by sending instruction message to *FireWall*'s control plane channel

$SDPSvrCfgFw(Acl, op) \triangleq$

$\land FwCtlChannel' = Append(FwCtlChannel, [Rule \mapsto Acl, op \mapsto op])$

$SDPSvrProcSpaAuthEx \triangleq$

$\land SDPSvrState = \text{“Work”}$

$\land AuthChannel \neq \langle\rangle$

$\land Head(AuthChannel).MsgID = \text{“SPA\_AUTH”}$ check the packet is SPA message

$\land Head(AuthChannel).dIP = SDPSvrInfo.IP$

$\land Head(AuthChannel).dPort = SDPSvrInfo.Port$

$\land \text{IF } FindAntiReplay(Head(AuthChannel), SDPSucSession) = \text{TRUE}$ case 1: the packet is a replay message

THEN

$\land SDPSvrAntiReplayAtk$ drop packets and record exception into log

$\land \text{UNCHANGED } user\_vars$

$\land$ UNCHANGED $\langle SDPSvrState, SDPSucSession, Account, SDPSvrInfo, SpoofCount, SpoofS$

$\land$ UNCHANGED fw_vars

$\land$ UNCHANGED $attacker\_vars$

$\land$ UNCHANGED $server\_vars$

$\land$ UNCHANGED $\langle uChannel, FwCtlChannel, FwDataChannel, aChannel, sChannel\rangle$

ELSE

$\land$ IF $SpaProcAuth(Head(AuthChannel), Account) = $ FALSE case 2: it is a spoof message or from

THEN

$\land$ $SDPSvrAntiSpoof$ drop packets and record exception into log

$\land$ UNCHANGED $user\_vars$

$\land$ UNCHANGED $\langle SDPSvrState, SDPSucSession, Account, SDPSvrInfo, ReplayCount,$

$\land$ UNCHANGED fw_vars

$\land$ UNCHANGED $attacker\_vars$

$\land$ UNCHANGED $server\_vars$

$\land$ UNCHANGED $\langle uChannel, FwCtlChannel, FwDataChannel, aChannel, sChannel\rangle$

ELSE case 3: Authenticated successfully, then send instruction to FW to allow data access towards ta

$\land$ $SDPSvrCfgFw([sIP \mapsto DeCrypt(Head(AuthChannel).CliIP, GetKey(Head(Aut$

$sPort \mapsto DeCrypt(Head(AuthChannel).CliPort, GetKey(Head(A$

$dIP \mapsto DeCrypt(Head(AuthChannel).SvrIP, GetKey(Head(Aut$

$dPort \mapsto DeCrypt(Head(AuthChannel).SvrPort, GetKey(Head(A$

$protocol \mapsto$ "TCP",

$action \mapsto$ "Accept"],

"Add" The instruction code is to Add a new rule.

)

$\land$ $SDPSucSession' = SDPSucSession \cup \{Head(AuthChannel)\}$ record in log

$\land$ $AuthChannel' = Tail(AuthChannel)$

$\land$ UNCHANGED $user\_vars$

$\land$ UNCHANGED $\langle SDPSvrState, Account, SDPSvrInfo, ReplayCount, SpoofCount, Repl$

$\land$ UNCHANGED fw_vars

$\land$ UNCHANGED $attacker\_vars$

$\land$ UNCHANGED $server\_vars$

$\land$ UNCHANGED $\langle uChannel, FwDataChannel, aChannel, sChannel\rangle$

Init state description of FireWall

Fire wall init: power on and enter work state, by default, it works in deny mode and will drop any ingress data packets.

$$FwInit \triangleq \land FwCtlChannel = \langle\rangle \\ \land FwDataChannel = \langle\rangle \\ \land FwState = \text{“Work”} \\ \land AclRuleSet = \{\} \\ \land AgedRuleSet = \{\} \\ \land DropPackets = \{\}$$

Next state actions of FireWall

Action 4: *FwProcAclCfg*

FireWall receive *Acl Rule* config instruction from control plane channel, and hence create a 3 Tuple rule for data access

Variables changed: $< FwCtlChannel,\ AclRuleSet >$

$FwProcAclCfg \triangleq$

$\land FwState = \text{“Work”}$

$\land FwCtlChannel \neq \langle\rangle$

$\land Head(FwCtlChannel).op = \text{“Add”}$ Check instruction message format

$\land AclRuleSet' = AclRuleSet \cup \{Head(FwCtlChannel).Rule\}$ Update local maintained rule table

$\land FwCtlChannel' = Tail(FwCtlChannel)$

$\land$ UNCHANGED $user\_vars$

$\land$ UNCHANGED $sdpsvr\_vars$

$\land$ UNCHANGED $attacker\_vars$

$\land$ UNCHANGED $server\_vars$

$\land$ UNCHANGED $\langle FwState,\ AgedRuleSet,\ DropPackets\rangle$

$\land$ UNCHANGED $\langle uChannel,\ AuthChannel,\ FwDataChannel,\ aChannel,\ sChannel\rangle$

Action 5: *FwProcEndPointAccessEx*

FireWall receive a ingress data packet from end point side and implement filtering function according to configed 4 Tuple *Acl Rule*.

For the improved version, the Firewall only confied with 4 tuples *ACL* Rules and *ACL Rule* automatically create function is prohibit

Variables changed: $< sChannel,\ AclRuleSet, FwDataChannel, DropPackets >$

Whether the *TCP* packet match a given 4 tuple rule.

$AclMatch4Tuple(p,\ Acl) \triangleq$

$\exists\, r \in Acl : (\wedge\, p.sIP = r.sIP$ $\quad$ $(sIP, sPort, dIP, dPort)$ must match exactly

$\qquad\qquad \wedge\, p.dIP = r.dIP$

$\qquad\qquad \wedge\, r.sPort \neq MATCH\_ANY$

$\qquad\qquad \wedge\, r.sPort = p.sPort$

$\qquad\qquad \wedge\, p.dPort = r.dPort$

$\qquad\qquad \wedge\, r.action = \text{“Accept”})$

$FwProcEndPointAccessEx \triangleq$

$\wedge\, FwState = \text{“Work”}$

$\wedge\, FwDataChannel \neq \langle\rangle$

$\wedge\, (\vee\, Head(FwDataChannel).Flg = \text{“TCP\_SYN”}$ to simplify the model, we only consider TCP connecti

$\quad \vee\, Head(FwDataChannel).Flg = \text{“TCP\_ACK”}$ the end point euipments as TCP client, only send TCP

$\quad)$

$\wedge\, (\text{IF } AclMatch4Tuple(Head(FwDataChannel), AclRuleSet)$

THEN $\quad$ $CASE1$: the incoming packets exactly match a 4 tuple rule

$\quad \wedge\, sChannel' = Append(sChannel, Head(FwDataChannel))$ route the packets to target server

$\quad \wedge\, FwDataChannel' = Tail(FwDataChannel)$

$\quad \wedge\, AclRuleSet' = AclRuleSet$

$\quad \wedge\, DropPackets' = DropPackets$

ELSE $\quad$ $CASE2$: the incoming packetsnot match any 4 tuple rule

$\quad \wedge\, FwDataChannel' = Tail(FwDataChannel)$

$\quad \wedge\, AclRuleSet' = AclRuleSet$

$\quad \wedge\, sChannel' = sChannel$ $\quad$ just drop the packets

$\quad \wedge\, DropPackets' = DropPackets \cup \{Head(FwDataChannel)\}$ record it into exception log

$\quad)$

$\wedge$ UNCHANGED $user\_vars$

$\wedge$ UNCHANGED $sdpsvr\_vars$

$\wedge$ UNCHANGED $attacker\_vars$

$\wedge$ UNCHANGED $\langle FwState, AgedRuleSet\rangle$

$\wedge$ UNCHANGED $server\_vars$

$\wedge$ UNCHANGED $\langle uChannel, AuthChannel, FwCtlChannel, aChannel\rangle$

Init state description of target service server

Target TCP server init and begin listening on its service IP and $Port$.

$$ServerInit \triangleq \land sState = \text{"Listen"}$$
$$\land sSvrInfo = [IP \mapsto SvrCfg.IP,\ Port \mapsto SvrCfg.Port] \quad \text{Load configuration}$$
$$\land sTCPLinkSet = \{\}$$
$$\land sChannel = \langle\rangle$$

Next state actions of target service server

Action 6: $ServerRcvTCPSyn$

Target server recieve a $TCP\ SYN$ packet from client side and try to allocate a new TCB.

Because the Firewall dose not filter server to endpoint direction packets, so to simplify the model, the server direcly sent $TCP\ ACK$

$uChannel$.

Variables changed: $< sTCPLinkSet, sChannel, uChannel, aChannel >$

Whether the coming packet indicates a new connection

$NewLink(p,\ LinkSet) \triangleq$

IF $LinkSet = \{\}$

THEN

TRUE

ELSE

(IF $\forall x \in LinkSet : ($ without matching TCB (TCP Control Block)

$\lor x.sIP \neq p.sIP$

$\lor x.dIP \neq p.dIP$

$\lor x.sPort \neq p.sPort$

$\lor x.dPort \neq p.dPort)$

THEN

TRUE

ELSE

FALSE)

$ServerRcvTCPSyn \triangleq$

$\land sState = \text{"Listen"}$

$\land sChannel \neq \langle\rangle$

$\land Head(sChannel).Flg = \text{"TCP\_SYN"}$

$\land Head(sChannel).dIP = sSvrInfo.IP$ check incoming packets format

$\land$ $Head(sChannel).dPort = sSvrInfo.Port$

$\land$ $sChannel' = Tail(sChannel)$

$\land$ (IF $NewLink(Head(sChannel), sTCPLinkSet)$

THEN $CASE1$: New TCP SYN packets

$\land$ $sTCPLinkSet' = sTCPLinkSet \cup \{$ create a TCB and update local link set.

$[dIP \mapsto Head(sChannel).sIP,$

$dPort \mapsto Head(sChannel).sPort,$

$sIP \mapsto Head(sChannel).dIP,$

$sPort \mapsto Head(sChannel).dPort,$

$Type \mapsto Head(sChannel).Type,$

$State \mapsto$ "SYN_RCVD" the TCB 's state is SYN_RCVD

$]\}$

$\land$ (IF $Head(sChannel).Type =$ "User"

THEN If the client is legal user, then send TCP_SYN_ACK packet to legal user.

($\land$ $uChannel' = Append(uChannel, [$

$sIP \mapsto Head(sChannel).dIP,$

$sPort \mapsto Head(sChannel).dPort,$

$dIP \mapsto Head(sChannel).sIP,$

$dPort \mapsto Head(sChannel).sPort,$

$Flg \mapsto$ "TCP_SYN_ACK",

$Type \mapsto Head(sChannel).Type]$

)

$\land$ $aChannel' = aChannel$

)

ELSE If the client is attacker, then send TCP_SYN_ACK packet to attacker.

($\land$ $aChannel' = Append(aChannel, [$

$sIP \mapsto Head(sChannel).dIP,$

$sPort \mapsto Head(sChannel).dPort,$

$dIP \mapsto Head(sChannel).sIP,$

$dPort \mapsto Head(sChannel).sPort,$

$Flg \mapsto$ "TCP_SYN_ACK",

$Type \mapsto Head(sChannel).Type]$

)

$\wedge\ uChannel' = uChannel$

$)$

$)$

ELSE *CASE2 :* duplicated *TCP SYN* packet,just neglect it for we don't focus on *TCP SYN* Flood attack.

$\wedge\quad sTCPLinkSet' = sTCPLinkSet$

$\wedge\quad aChannel' = aChannel$

$\wedge\quad uChannel' = uChannel$

$)$

$\wedge$ UNCHANGED $user\_vars$

$\wedge$ UNCHANGED $sdpsvr\_vars$

$\wedge$ UNCHANGED $attacker\_vars$

$\wedge$ UNCHANGED $\langle sState, sSvrInfo\rangle$

$\wedge$ UNCHANGED fw_vars

$\wedge$ UNCHANGED $\langle AuthChannel, FwCtlChannel, FwDataChannel\rangle$

Action 7: *ServerRcvTcpAck*

Target server recieve a *TCP ACK* packet that acknowledge the last *SYN_ACK*, then establish the *TCP* link with the client.

Variables changed: $< sTCPLinkSet, sChannel >$

$ServerRcvTcpAck \triangleq$

$\wedge\ sState = \text{“Listen”}$

$\wedge\ sChannel \neq \langle\rangle$

$\wedge\ Head(sChannel).Flg = \text{“TCP\_ACK”}$ check incoming packets format

$\wedge\ HasMatchLink(Head(sChannel), sTCPLinkSet)$

$\wedge\ GetMatchLink(Head(sChannel), sTCPLinkSet).State = \text{“SYN\_RCVD”}$ the matched *TCB* state must be *SYN*

$\wedge\ sChannel' = Tail(sChannel)$

$$\begin{aligned}\wedge\ sTCPLinkSet' = (sTCPLinkSet \setminus \{GetMatchLink(Head(sChannel), sTCPLinkSet)\})\\ \cup \{[sIP &\mapsto GetMatchLink(Head(sChannel), sTCPLinkSet).sIP,\\ sPort &\mapsto GetMatchLink(Head(sChannel), sTCPLinkSet).sPort,\\ dIP &\mapsto GetMatchLink(Head(sChannel), sTCPLinkSet).dIP,\\ dPort &\mapsto GetMatchLink(Head(sChannel), sTCPLinkSet).dPort,\\ Type &\mapsto GetMatchLink(Head(sChannel), sTCPLinkSet).Type,\\ State &\mapsto \text{“ESTABLISHED”}\\ ]\end{aligned}$$

Update *TCP* link state to *ESTABLISHED*.

This indicates the client has successfully accessed target server

$\}$

$\land$ UNCHANGED $user\_vars$

$\land$ UNCHANGED $sdpsvr\_vars$

$\land$ UNCHANGED $attacker\_vars$

$\land$ UNCHANGED $\langle sState, sSvrInfo\rangle$

$\land$ UNCHANGED fw_vars

$\land$ UNCHANGED $\langle uChannel, AuthChannel, FwCtlChannel, FwDataChannel, aChannel\rangle$

Init state description of Attacker

Attacker init and capable of sniffing the packets on the local network.

$AttackerInit \triangleq \land aState =$ "Listen"
$\land AuthKnowledge = \{\}$
$\land aSession = \{\}$
$\land aTCPLinkSet = \{\}$
$\land aChannel = \langle\rangle$
$\land sniffCount = 0$
$\land CapAuthMsg = \{\}$
$\land aCounter = 0$
$\land aIP = AttackerCfg.SrcIp$
$\land DataKnowledge = \{\}$
$\land CapDataMsg = \{\}$

Next state actions of attacker

Action 8: *AttackerSniffAuthChannel*

Attacker eavesdropping SPA message from legal user to *SDP* controller by sniffing the *Auth* channel.

Once a new SPA message is captured,attacker will duplicate it into its current Auth-knowledge set.

We don't guarantee every new SPA message can be captured by attacker, it only has the opportuity to get each messag

Variables changed: $< AuthKnowledge, CapAuthMsg, sniffCount >$

Slect a new (which means unknown to attacker till now) SPA message from the *Auth* channel

to simulate a successful sniff.

$SelectNewAuthMsg(MsgQ, known) \triangleq$

IF $known \neq \{\}$

THEN for a dedicate user, the difference among SPA messages is the value of *SN* (counter) field.

CHOOSE $S \in$ SUBSET $Seq2Set(MsgQ) : (\forall x \in S : (\forall y \in known : x.Tstamp \neq y.Tstamp))$

ELSE

$Seq2Set(MsgQ)$

For the attacker can also insert fake messages into channel, but

for both data and auth channel, attacker only wants to capture messages from legal user.

so the $PureChannel()$ function is to select the set of user's messages.

$PureChannel(S) \triangleq SelectSeq(S,$ LAMBDA $x : x.Type =$ "User"$)$

$AttackerSniffAuthChannel \triangleq$

$\wedge\ aState =$ "Listen"

$\wedge\ PureChannel(AuthChannel) \neq \langle\rangle$ pre-condition: there exists attacker unknown legal user originated SPA messages on t

$\wedge\ \exists\, i \in 1 \,..\, Len(PureChannel(AuthChannel)) :$

$(\forall x \in CapAuthMsg : PureChannel(AuthChannel)[i].Tstamp \neq x.Tstamp)$

$\wedge\ AuthKnowledge' = AuthKnowledge \cup$ post-condition: attacker learned new intelligence by a successful sniffing.

$SelectNewAuthMsg(PureChannel(AuthChannel), CapAuthMsg)$

$\wedge\ CapAuthMsg' = CapAuthMsg \cup$ All the captured message in history recorded in Log.

$SelectNewAuthMsg(PureChannel(AuthChannel), CapAuthMsg)$

$\wedge\ sniffCount' = sniffCount + 1$ increase statistics

$\wedge$ UNCHANGED $user\_vars$

$\wedge$ UNCHANGED $sdpsvr\_vars$

$\wedge$ UNCHANGED fw_vars

$\wedge$ UNCHANGED $server\_vars$

$\wedge$ UNCHANGED $\langle aState, aSession, aTCPLinkSet, aCounter, aIP, DataKnowledge, CapDataMsg \rangle$

$\wedge$ UNCHANGED $Public\_vars$

Action 9: $AttackerSniffDataChannel$

Attacker eavesdropping data access from legal user to target server by sniffing the data channel.

Once a new data packet is captured,it will duplicate it into its current data-knowledge set.

We don't guarantee every new data packets can be captured by attacker, it only has the opportuity to get each packets.

Variables changed: $< DataKnowledge, CapDataMsg >$

Slect a new (which means unknown to attacker till now) data packets being sent from user to $FireWall$

to simulate a successful sniff.

$SelectNewDataMsg(MsgQ, known) \triangleq$

IF $known \neq \{\}$

THEN The aim of capturing user data access packets is to get the exposure service info about the target serve

so (dIP,$dPort$) is the key knowledge.

CHOOSE $S \in$ SUBSET $Seq2Set(MsgQ) : (\forall\, x \in S : (\forall\, y \in known : (x.dIP \neq y.dIP \wedge x.dPort \neq y$

ELSE

$Seq2Set(MsgQ)$

$AttackerSniffDataChannel \triangleq$

$\wedge\ aState =$ "Listen"

$\wedge\ PureChannel(FwDataChannel) \neq \langle\rangle$ pre-condition: there exists attacker unknown target server service i

$\wedge\ \exists\, i \in 1\,..\,Len(PureChannel(FwDataChannel)) :$

$(\forall\, x \in CapDataMsg :\ \wedge\ PureChannel(FwDataChannel)[i].dIP \neq x.dIP$

$\wedge\ PureChannel(FwDataChannel)[i].dPort \neq x.dPort$

$\wedge\ PureChannel(FwDataChannel)[i].Flg =$ "TCP_SYN" A new TCP

)

$\wedge\ DataKnowledge' = DataKnowledge\ \cup$ post-condition: attacker learned new intelligence by a successful

$SelectNewDataMsg(PureChannel(FwDataChannel),\ CapDataMsg)$

$\wedge\ CapDataMsg' = CapDataMsg\ \cup$ All the captured packets in history recorded in Log.

$SelectNewDataMsg(PureChannel(FwDataChannel),\ CapDataMsg)$

$\wedge\ sniffCount' = sniffCount + 1$ increase statistics

$\wedge$ UNCHANGED $user\_vars$

$\wedge$ UNCHANGED $sdpsvr\_vars$

$\wedge$ UNCHANGED fw_vars

$\wedge$ UNCHANGED $server\_vars$

$\wedge$ UNCHANGED $\langle aState,\ AuthKnowledge,\ \ aSession,\ aTCPLinkSet,\ CapAuthMsg,\ aCounter,\ a$

$\wedge$ UNCHANGED $Public\_vars$

Action 10: $AttackerSpoofAuth$

Attacker build and send fake SPA messages to SDP controller by spoofing legal user.

The making of each fake message is based on one corrsponding element in the Auth-Knowledge set, one element in the

can only be used to produce one spoof message.

The spoof message re-use the legal user's ID and all other fields except SN ($Tstamp$) field increasing to avoid anti-rep

Variables changed: $< aSession, AuthChannel, AuthKnowledge >$

make a spoof message according a captured auth knowledge

$SpoofAuthMsg(m) \triangleq$

$[MsgID \mapsto$ "SPA_AUTH",
$sIP \mapsto m.sIP,$
$sPort \mapsto m.sPort,$
$dIP \mapsto m.dIP,$
$dPort \mapsto m.dPort,$
$ClientID \mapsto m.ClientID,$
$Tstamp \mapsto m.Tstamp + 1,$ *SN* number increase
$CliIP \mapsto m.CliIP,$
$CliPort \mapsto m.CliPort,$
$SvrIP \mapsto m.SvrIP,$
$SvrPort \mapsto m.SvrPort,$
$HMAC \mapsto m.HMAC,$
$Type \mapsto$ "Attacker"$]$

$AttackerSpoofAuth \triangleq$

$\land AuthKnowledge \neq \{\}$ pre-condition: there exists intellicence about user's auth message learned by sniffing.
$\land AuthChannel' = Append(AuthChannel, SpoofAuthMsg(\text{CHOOSE } x \in AuthKnowledge : \text{TRUE}))$ send new bui
$\land aSession' = aSession \cup \{SpoofAuthMsg(\text{CHOOSE } x \in AuthKnowledge : \text{TRUE})\}$ new Attack session is recorded
$\land AuthKnowledge' = AuthKnowledge \setminus \{\text{CHOOSE } x \in AuthKnowledge : \text{TRUE}\}$ one knowledge item can be only be
$\land$ UNCHANGED $user\_vars$
$\land$ UNCHANGED $sdpsvr\_vars$
$\land$ UNCHANGED fw_vars
$\land$ UNCHANGED $server\_vars$
$\land$ UNCHANGED $\langle aState, aTCPLinkSet, sniffCount, CapAuthMsg, aCounter, aIP, DataKnowledge, CapDat$
$\land$ UNCHANGED $\langle uChannel, FwCtlChannel, FwDataChannel, aChannel, sChannel\rangle$

Action 11: *AttackerReplayAuth*

Attacker build and send fake SPA messages to *SDP* controller by replay legal user's message.

The making of each fake message is based on one corrsponding element in the Auth-Knowledge set, one element in the knowledge set can only be used to produce one replay message.

Variables changed: $< aSession, AuthChannel, AuthKnowledge >$

$ReplayAuthMsg(m) \triangleq$ make replay message by duplication.

$$
\begin{aligned}
[&MsgID \mapsto \text{“SPA\_AUTH”},\\
&sIP \mapsto m.sIP,\\
&sPort \mapsto m.sPort,\\
&dIP \mapsto m.dIP,\\
&dPort \mapsto m.dPort,\\
&ClientID \mapsto m.ClientID,\\
&Tstamp \mapsto m.Tstamp,\\
&CliIP \mapsto m.CliIP,\\
&CliPort \mapsto m.CliPort,\\
&SvrIP \mapsto m.SvrIP,\\
&SvrPort \mapsto m.SvrPort,\\
&HMAC \mapsto m.HMAC,\\
&Type \mapsto \text{“Attacker”}]
\end{aligned}
$$

$AttackerReplayAuth \triangleq$

$\land\ AuthKnowledge \neq \{\}$ pre-condition: there exists intellicence about user's auth message learned by sniffing.

$\land\ AuthChannel' = Append(AuthChannel,\ ReplayAuthMsg(\text{CHOOSE } x \in AuthKnowledge : \text{TRUE}$

$\land\ aSession' = aSession \cup \{ReplayAuthMsg(\text{CHOOSE } x \in AuthKnowledge : \text{TRUE})\}$ new Attack ses

$\land\ AuthKnowledge' = AuthKnowledge \setminus \{\text{CHOOSE } x \in AuthKnowledge : \text{TRUE}\}$ one knowledge item

$\land$ UNCHANGED $user\_vars$

$\land$ UNCHANGED $sdpsvr\_vars$

$\land$ UNCHANGED fw_vars

$\land$ UNCHANGED $server\_vars$

$\land$ UNCHANGED $\langle aState,\ aTCPLinkSet,\ sniffCount,\ CapAuthMsg,\ aCounter,\ aIP,\ DataKnowled$

$\land$ UNCHANGED $\langle uChannel,\ FwCtlChannel,\ FwDataChannel,\ aChannel,\ sChannel\rangle$

Action 12: *AttackerBrutalAttck*

Attacker try to brutally connect the target server only by the intelligence got from user's *Auth* message.

The making of each tcp connection is based on one auth atttack session, one element in the history auth attack session

can only be used to produce one brutal attack message.

Variables changed: $< aSession, AuthChannel, AuthKnowledge, FwDataChannel >$

$AttckerBulidTcpSynPktByAuthMsg(m) \triangleq$ attack try to connect target service server as a TCP client, send SYN packet in

$[sIP \mapsto aIP,$

$sPort \mapsto RandomPort(aCounter, ATTACKER\_BASEPORT),$ local port generated randomly

$dIP \mapsto m.SvrIP,$ target server info directly get from previously auth message m.

$dPort \mapsto m.SvrPort,$

$Flg \mapsto$ "TCP_SYN",

$Type \mapsto$ "Attacker"]

$Exist\_aSession4Battck \triangleq$

$\exists x \in aSession : (\forall y \in aTCPLinkSet : x.Tstamp \neq y.AuthID)$

$Get\_aSession4Battck \triangleq$ choose an historic auth attack session to make a brutal data access attack

CHOOSE $x \in aSession : (\forall y \in aTCPLinkSet : x.Tstamp \neq y.AuthID)$

$AttackerBrutalAttck \triangleq$

$\land\ aSession \neq \{\}$

$\land\ (\lor\ aTCPLinkSet = \{\}$

$\lor\ (\land\ aTCPLinkSet \neq \{\}$ pre-condition: there exists at least one auth attack session without brutal attack had happene

$\land\ Exist\_aSession4Battck$

$)$

$)$

$\land\ aCounter' = aCounter + 1$ acounter is used to build the local port value of the TCP connection, increase each time to a

$\land\ FwDataChannel' = Append(FwDataChannel, AttckerBulidTcpSynPktByAuthMsg(Get\_aSession4Battck))$

$\land\ aTCPLinkSet' = aTCPLinkSet \cup \{$ maintain local TCP socket

$[sIP \mapsto AttckerBulidTcpSynPktByAuthMsg(Get\_aSession4Battck).sIP,$

$sPort \mapsto AttckerBulidTcpSynPktByAuthMsg(Get\_aSession4Battck).sPort,$

$dIP \mapsto AttckerBulidTcpSynPktByAuthMsg(Get\_aSession4Battck).dIP,$

$dPort \mapsto AttckerBulidTcpSynPktByAuthMsg(Get\_aSession4Battck).dPort,$

$State \mapsto$ "SYN_SENT", the tcp link's state now is SYN_SENT

$AuthID \mapsto Get\_aSession4Battck.Tstamp$ this field is used to relate to the corresponding auth attack session.

$]\ \}$

$\land$ UNCHANGED $user\_vars$

$\land$ UNCHANGED $sdpsvr\_vars$

$\land$ UNCHANGED fw_vars

$\wedge$ UNCHANGED *server_vars*

$\wedge$ UNCHANGED $\langle aState, AuthKnowledge, aSession, sniffCount, CapAuthMsg, aIP, DataKnowle$

$\wedge$ UNCHANGED $\langle uChannel, AuthChannel, FwCtlChannel, aChannel, sChannel \rangle$

Action 13: *AttackerInspectSvr*

Attacker try to connect target server according to intelligence of previously captured data plane traffic info by sending

The making of each tcp connection is based on one element in the Data Knowledge set which is learned by sniffing lega

with target server.

one knowledge can only be used to produce one inspection attempt.

Variables changed: $< aCounter, FwDataChannel, aTCPLinkSet, DataKnowledge >$

$AttckerBulidTcpSynPktByData(p) \triangleq$

$[sIP \mapsto aIP,$

$sPort \mapsto RandomPort(aCounter, ATTACKER\_BASEPORT),$

$dIP \mapsto p.dIP,$

$dPort \mapsto p.dPort,$

$Flg \mapsto$ "TCP_SYN",

$Type \mapsto$ "Attacker"]

$AttackerInspectSvr \triangleq$

$\wedge DataKnowledge \neq \{\}$ pre-condition: there exists learned data knowledge that still not used to makea inspe

$\wedge aCounter' = aCounter + 1$ acounter is used to build the local port value of the *TCP* connection, increase

$\wedge FwDataChannel' = Append(FwDataChannel, AttckerBulidTcpSynPktByData($CHOOSE $x \in I$

$\wedge aTCPLinkSet' = aTCPLinkSet \cup \{$ maintain local *TCP* socket

$[sIP \mapsto AttckerBulidTcpSynPktByData($CHOOSE $x \in DataKnowledge :$ TRUE$).sIP,$

$sPort \mapsto AttckerBulidTcpSynPktByData($CHOOSE $x \in DataKnowledge :$ TRUE$).sPort,$

$dIP \mapsto AttckerBulidTcpSynPktByData($CHOOSE $x \in DataKnowledge :$ TRUE$).dIP,$

$dPort \mapsto AttckerBulidTcpSynPktByData($CHOOSE $x \in DataKnowledge :$ TRUE$).dPort,$

$State \mapsto$ "SYN_SENT", the tcp link's state now is *SYN_SENT*

$AuthID \mapsto UNKNOWN\_AUTH\_ID$ This tcp connection is built accroding to captured data plane

$] \}$

$\wedge DataKnowledge' = AuthKnowledge \setminus \{$CHOOSE $x \in DataKnowledge :$ TRUE$\}$ one knowledge item

$\wedge$ UNCHANGED *user_vars*

$\wedge$ UNCHANGED *sdpsvr_vars*

$\wedge$ UNCHANGED fw_vars

$\wedge$ UNCHANGED $server\_vars$

$\wedge$ UNCHANGED $\langle aState, AuthKnowledge, aSession, sniffCount, CapAuthMsg, aIP, CapDataMsg\rangle$

$\wedge$ UNCHANGED $\langle uChannel, AuthChannel, FwCtlChannel, aChannel, sChannel\rangle$

Action 14: *AttackerRcvSynAck*

Attacker's inspection *TCP* connection estalished trigered by receiving *TCP SYN ACK* pakcet from target server.

This indicates the attacker fufilled a inspection attack to the target server.

Because the Firewall dose not filter server to endpoint direction packets, so to simplify the model, the server direcly sent *TCP* packe

uChannel or *aChannel* to simplify the model.

Variables changed: < *aTCPLinkSet*,*aChannel*,*FwDataChannel* >

$AttackerRcvSynAck \triangleq$

$\wedge\ aTCPLinkSet \neq \{\}$

$\wedge\ aChannel \neq \langle\rangle$

$\wedge\ Head(aChannel).Flg = \text{“TCP\_SYN\_ACK”}$

$\wedge\ Head(aChannel).Type = \text{“Attacker”}$

$\wedge\ HasMatchLink(Head(aChannel), aTCPLinkSet)$

$\wedge\ GetMatchLink(Head(aChannel), aTCPLinkSet).State = \text{“SYN\_SENT”}$ pre-condition: local *TCP* client in the

$\wedge\ aTCPLinkSet' = (aTCPLinkSet \setminus \{GetMatchLink(Head(aChannel), aTCPLinkSet)\})$ Post-condition: The

$$\cup \{[sIP \mapsto GetMatchLink(Head(aChannel), aTCPLinkSet).sIP,$$
$$sPort \mapsto GetMatchLink(Head(aChannel), aTCPLinkSet).sPort,$$
$$dIP \mapsto GetMatchLink(Head(aChannel), aTCPLinkSet).dIP,$$
$$dPort \mapsto GetMatchLink(Head(aChannel), aTCPLinkSet).dPort,$$
$$State \mapsto \text{“ESTABLISHED”},$$
$$AuthID \mapsto GetMatchLink(Head(aChannel), aTCPLinkSet).AuthID$$
$$]$$
$$\}$$

$\wedge\ aChannel' = Tail(aChannel)$

$\wedge\ FwDataChannel' = Append(FwDataChannel, EndPointBulidTcpAckPkt(Head(aChannel), \text{“Attacker”}))$

$\wedge$ UNCHANGED $user\_vars$

$\wedge$ UNCHANGED $sdpsvr\_vars$

$\wedge$ UNCHANGED fw_vars

$\wedge$ UNCHANGED $\langle aState, AuthKnowledge, aSession, sniffCount, CapAuthMsg, aCounter, aIP, DataKnowl$

$\wedge$ UNCHANGED $server\_vars$
$\wedge$ UNCHANGED $\langle uChannel, AuthChannel, FwCtlChannel, sChannel\rangle$

The init description of the whole system

$Init \triangleq$ $\wedge$ $UsrInit$
$\wedge$ $SDPSvrInit$
$\wedge$ $FwInit$
$\wedge$ $AttackerInit$
$\wedge$ $ServerInit$

Next state transtion of the whole system

The next state actions of the whole system is the disjunction of each entity's next state action.

$Next \triangleq$ User's next state actions
$\vee$ $UsrConnectServerEnhance$
$\vee$ $UsrRcvSynAck$
$\vee$ $UsrReConnectServer$
SDP controller's next state actions
$\vee$ $SDPSvrProcSpaAuthEx$
Fire Wall's next state actions
$\vee$ $FwProcAclCfg$
$\vee$ $FwProcEndPointAccessEx$
Attacker's next state actions
$\vee$ $AttackerSniffAuthChannel$
$\vee$ $AttackerSpoofAuth$
$\vee$ $AttackerReplayAuth$
$\vee$ $AttackerBrutalAttck$
$\vee$ $AttackerSniffDataChannel$
$\vee$ $AttackerInspectSvr$
$\vee$ $AttackerRcvSynAck$
Target service server's next state actions
$\vee$ $ServerRcvTCPSyn$
$\vee$ $ServerRcvTcpAck$

The specification of the whole system

$Spec \triangleq Init \land \Box[Next]_{vars}$

The Fair specification of the whole system

$FairSpec \triangleq$ *WF* means weak fairness, gurantee once the action is enabled, it will be triggered sooner or later.

$\land\ Spec$ Use the fairness attribute to eliminate unnecessary stuttering states.

$\land\ \mathrm{WF}_{vars}(UsrConnectServerEnhance)$

$\land\ \mathrm{WF}_{vars}(UsrRcvSynAck)$

$\land\ \mathrm{WF}_{vars}(UsrReConnectServer)$

$\land\ \mathrm{WF}_{vars}(SDPSvrProcSpaAuthEx)$

$\land\ \mathrm{WF}_{vars}(FwProcAclCfg)$

$\land\ \mathrm{WF}_{vars}(AttackerSniffAuthChannel)$

$\land\ \mathrm{WF}_{vars}(AttackerSpoofAuth)$

$\land\ \mathrm{WF}_{vars}(AttackerReplayAuth)$

$\land\ \mathrm{WF}_{vars}(FwProcEndPointAccessEx)$

$\land\ \mathrm{WF}_{vars}(ServerRcvTCPSyn)$

$\land\ \mathrm{WF}_{vars}(ServerRcvTcpAck)$

$\land\ \mathrm{WF}_{vars}(AttackerBrutalAttck)$

$\land\ \mathrm{WF}_{vars}(AttackerSniffDataChannel)$

$\land\ \mathrm{WF}_{vars}(AttackerInspectSvr)$

$\land\ \mathrm{WF}_{vars}(AttackerRcvSynAck)$

Invariants to be verified

$DataAccessSafeLaw \triangleq$ attacker can not find target server service at anytime

$\land\ \forall x \in aTCPLinkSet : x.State \neq$ "ESTABLISHED"

$SPASafeLaw \triangleq$ attacker can not launch a successful SPA auth at anytime

$\land\ \forall x \in SDPSucSession : x.Type \neq$ "Attacker"

The temporal properties of the system to be verified

Temporal Property 1: *SPA_AvailableProperty*

This formula asserts the availability of SPA service provided by the *SDP* controller

$AuthMessageMatch(m, n) \triangleq$ Both m and n are auth Sessions

$\land\ m.MsgID = n.MsgID$

$\land\ m.sIP = n.sIP$

$\land\ m.sPort = n.sPort$

$\land\ m.dIP = n.dIP$

$\land\ m.dPort = n.dPort$

$\land\ m.ClientID = n.ClientID$

$\land\ m.Tstamp = n.Tstamp$

$\land\ m.SvrIP = n.SvrIP$

$\land\ m.SvrPort = n.SvrPort$

$\land\ m.HMAC = n.HMAC$

$\land\ m.Type = n.Type$

$SDP\_AclRuleMatchAuth(m,\ r) \triangleq$ m is an auth Session, r is a *ACL Rule*

$\land\ DeCrypt(m.CliIP,\ GetKey(m.ClientID,\ Account)) = r.sIP$

$\land\ DeCrypt(m.CliPort,\ GetKey(m.ClientID,\ Account)) = r.sPort$

$\land\ DeCrypt(m.SvrIP,\ GetKey(m.ClientID,\ Account)) = r.dIP$

$\land\ DeCrypt(m.SvrPort,\ GetKey(m.ClientID,\ Account)) = r.dPort$

$\land\ r.protocol = $ "TCP"

$\land\ r.action = $ "Accept"

This formula asserts that the system's behavior eventually always meets the underlying propositions

1. All authentication sessions launched by legal users have been successfully processed by *SDP* controller.
2. All successfully processed *Auth* sessions recorded by *SDP* controller are sessions launched by legal users.
3. For all sucessfully authenticated sessions, the Fire wall has been configed corresponding *ACL Rule*.

$SPA\_AvailableProperty \triangleq$

$\Diamond\Box(\ \land\ \forall\, x \in uAuthSession : (\exists\, y \in SDPSucSession : AuthMessageMatch(x,\ y))$ user $\rightarrow$ contro

$\land\ \forall\, x \in SDPSucSession : (\exists\, y \in uAuthSession :\ AuthMessageMatch(x,\ y))$ controller $\rightarrow$ u

$\land\ \forall\, x \in uAuthSession : (\exists\, y \in (AclRuleSet \cup AgedRuleSet) : SDP\_AclRuleMatchAuth(x,$

$)$

Temporal Property 2: *SPA_AntiDosProperty*

This formula asserts the Anti-Dos property of *SDP* controller,which means the controller can always inspect and defeat spoof and replay attack.

The following formula asserts that every SPA replay attack inspected by the *SDP* controller is originated from the atta

$SPA\_AntiReplayProperty \triangleq$

IF $ReplaySession \neq \{\}$

THEN

$\forall\, x \in ReplaySession : (\exists\, y \in aSession : AuthMessageMatch(x,\, y))$

ELSE

TRUE

The following formula asserts that every SPA spoof attack inspected by the *SDP* controller is originated from the attacker

$SPA\_AntiSpoofProperty \triangleq$

IF $SpoofSession \neq \{\}$

THEN

$\forall\, x \in SpoofSession : (\exists\, y \in aSession : AuthMessageMatch(x,\, y))$

ELSE

TRUE

The following formula asserts that the system's behavior eventually always meets the underlying propositions

IF attacker ever captured legal SPA packets by sniffing,then:

1. For every captured legal SPA messages, the attacker will launch a SPA attack according to the message info.
2. Every SPA attack message launched by the attacker will be inspected and blocked by the *SDP* controller.

IF attacker never captured legal SPA packets, then no SPA attack is lanched.

$SPA\_AntiDosProperty \triangleq$

$\Diamond\Box(\ \wedge\ CapAuthMsg \subseteq uAuthSession$

$\wedge\ Cardinality(CapAuthMsg) = Cardinality(aSession)$

$\wedge$ IF $aSession \neq \{\}$

THEN

$\wedge\ \forall\, x \in aSession : (\exists\, y \in (ReplaySession \cup SpoofSession) : AuthMessageMatch(x,\, y))$

$\wedge\ SPA\_AntiReplayProperty$

$\wedge\ SPA\_AntiSpoofProperty$

ELSE

$\wedge\ CapAuthMsg = \{\}$

$\wedge\ ReplaySession = \{\}$

$\wedge\ SpoofSession = \{\}$

$)$

$$CliSvrLinkMatch(c,\ s)\ \triangleq$$
$$\land\ c.dIP = s.sIP$$
$$\land\ c.sIP\ = s.dIP$$
$$\land\ c.dPort = s.sPort$$
$$\land\ c.sPort\ = s.dPort$$

Temporal Property 3: *UserAccessAvailProperty*

This formula asserts the availability of the data plane service ,which means

legal user can finally access the target server except the case that 3 tuple *Acl Rule* is aged before th *TCP* connection e

$$UserAccessAvailProperty\ \triangleq$$
$$\Diamond\Box(\ \land (\forall\, x \in\ uTCPLinkSet :\ \lor (\ \land x.State = \text{"ESTABLISHED"}$$ *scenario*1: *TCP* link established, an
$$\land\ \exists\, y \in sTCPLinkSet : (CliSvrLinkMatch(x,\ y) \land x.State = y$$
$$\land\ AclMatch4Tuple(x,\ AclRuleSet)$$
$$)$$
$$)$$
$$\land\ uTCPLinkSet \neq \{\}$$
$$)$$

Temporal Property 4: *SvrHidenProperty*

This formula asserts the service hidden property of the *SDP* arhitecture. which means

finally attacker can not establish any link with the target server.

$$SvrHidenProperty\ \triangleq$$
$$\Diamond\Box(\ \land (\forall\, x \in sTCPLinkSet :\ \land x.Type \neq \text{"Attacker"}$$
$$\land\ x.State = \text{"ESTABLISHED"})$$ All the established link in server side are
$$\land (\forall\, y \in aTCPLinkSet :\ \land y.State \neq \text{"ESTABLISHED"})$$ Attacker as a *TCP* client, no established
$$)$$

Temporal Property 5: *FwRuleConsistentProperty*

This formula asserts that for each successful auth session in history there exists

a corresponding 3 Tuple *Acl Rule* on *FW*, available or aged, vice versa.

$$Get3TupleAclRuleSet(S)\ \triangleq$$

get all the 3 *Tpule Acl* rule in history.

$$\text{LET}\ \ CS[T \in \text{SUBSET}\ S]\ \triangleq\ \text{IF}\ \ T = \{\}$$

$$
\begin{array}{l}
\text{THEN } \{\} \\
\text{ELSE} \\
(\\
\text{IF } (\text{CHOOSE } x \in T : \text{TRUE}).sPort = MATCH\_ANY \\
\quad \text{THEN} \\
\{\text{CHOOSE } x \in T : \text{TRUE}\} \cup CS[T \setminus \{\text{CHOOSE } x \in T : \text{TRUE}\}] \\
\quad \text{ELSE} \\
CS[T \setminus \{\text{CHOOSE } x \in T : \text{TRUE}\}] \\
)
\end{array}
$$

$\text{IN} \quad CS[S]$

$$
\begin{array}{l}
AuthRelateAcl(s, R) \triangleq \\
\quad \exists\, r \in R : SDP\_AclRuleMatchAuth(s, r)
\end{array}
$$

$$
\begin{array}{l}
AclRelateAuth(r, S) \triangleq \\
\quad \exists\, s \in S : SDP\_AclRuleMatchAuth(s, r)
\end{array}
$$

$FwRuleConsistentProperty \triangleq$ * the consistent between user's SPA session and ever configed $L3$ tuple *Acl Rule* on Fire Wall

$$
\begin{array}{l}
\Diamond\Box\ (\quad \wedge\ Cardinality(uAuthSession) = Cardinality(Get3TupleAclRuleSet(AclRuleSet \cup AgedRuleSet)) \\
\qquad \wedge\ \forall\, x \in uAuthSession : AuthRelateAcl(x, Get3TupleAclRuleSet(AclRuleSet \cup AgedRuleSet)) \\
\qquad \wedge\ \forall\, y \in Get3TupleAclRuleSet(AclRuleSet \cup AgedRuleSet) : AclRelateAuth(y, uAuthSession) \\
\quad)
\end{array}
$$

$FwRuleConsistentProperty \triangleq$ the consistent between user's SPA session and ever configed $L3$ tuple *Acl Rule* on Fire Wall

$$
\begin{array}{l}
\Diamond\Box(\wedge\ Cardinality(uAuthSession) = Cardinality(AclRuleSet) \\
\qquad \wedge\ \forall\, x \in uAuthSession : AuthRelateAcl(x, AclRuleSet) \\
\qquad \wedge\ \forall\, y \in AclRuleSet : AclRelateAuth(y, uAuthSession) \\
\qquad \wedge\ AgedRuleSet = \{\} \\
\quad)
\end{array}
$$

Temporal Property 6: *FwCorrectProperty*

This formula asserts that the Fire Wall's Packets filitering function works well, which means that for any unestablished *TCP* links there must exists packets dropping by *FireWall*.

$WithOutDropPkts(x) \triangleq \neg WithDropPkts(x)$

$FwCorrectProperty \triangleq$ to simplify the model, we don't consider TCP packets re-transport mechanism for attacke
so attacker established TCP links without packet dropping.
But for a legal user, its first TCP SYN packets might be routed to the FW before the co
so commonly, this TCP link should be established by the retransmit of SYN packet after

$$
\begin{array}{l}
\Diamond\Box(\quad \land \forall x \in aTCPLinkSet : \text{ IF } x.State = \text{"ESTABLISHED"} \\
\qquad\qquad\qquad \text{THEN} \\
\qquad\qquad\qquad WithOutDropPkts(x) \\
\qquad\qquad\qquad \text{ELSE} \\
\qquad\qquad\qquad WithDropPkts(x) \\
\qquad \land \forall x \in uTCPLinkSet : \text{ IF } (x.State = \text{"ESTABLISHED"} \land x.Retrans = \text{FALSE}) \\
\qquad\qquad\qquad \text{THEN} \\
\qquad\qquad\qquad WithOutDropPkts(x) \\
\qquad\qquad\qquad \text{ELSE} \\
\qquad\qquad\qquad WithDropPkts(x) \\
\quad)
\end{array}
$$

Example:
Terminal $\triangleq \Diamond\Box($
\ * ˆ
)

\ * Modification History
\ * Last modified *Thu Feb* 24 15:38:05 *CST* 2022 by 10227694
\ * Created *Mon Jan* 17 16:29:42 *CST* 2022 by 10227694