Editors: Iván Arce, ivan.arce@corest.com
Elias Levy, aleph1@securityfocus.com

An Analysis of the Slapper Worm

During the past decade, security bugs' impact on a society dependent on a seamless and secure flow of information has become painfully evident. We've all learned the implications of security bugs and breaches the hard way, in a defensive and after-the-fact manner that prompts us to plug holes quickly and then wait for the next big one to surface. With the overwhelming amount of bug reports and security threats made public every day, it is daunting and difficult to identify trends and have a reasonable expectation of adopting a proactive information security strategy that deals with possible future threats.

IVÁN ARCE
Core Security Technologies

ELIAS LEVY
Symantec Corporation

The Attack Trends department attempts to examine security bugs and exploitation programs and techniques that might be useful to understand the current state of information security threats and their future; this analysis could help us improve our security intelligence.

For this premier issue of *IEEE Security & Privacy* magazine, we present an analysis of the Slapper worm, an automated attack tool that compromised thousands of servers and propagated across network and geographical boundaries on its own accord in September 2002 (see http://online.securityfocus.com/archive/1/291748/2002-09-07/2002-09-13/2). It has a rich mix of features for both a historical and a strictly technical analysis and can help us identify some expectations for future information security threats.

Opening a can of worms

On 2 November 1988, the Internet suffered a major attack. It was on the verge of collapse for several hours and in a critical state until 8 November 1988, when officials at the National Computer Security Center in Ft. Meade, Maryland assumed that the incident was over. The researchers determined that the culprit was a computer program.[1]

A few days after the attack's inception, while still suffering from network communication problems and overloaded computer systems, several computer scientists and security experts decompiled and analyzed the attacking program.

Three weeks later, Eugene Spafford published a detailed analysis of the program,[2] and the term *worm* was first used to describe the program's behavior:

> *"A worm is a program that can run by itself and can propagate a fully working version of itself to other machines. It is derived from the word tapeworm, a parasitic organism that lives inside a host and saps its resources to maintain itself."*[2]

The program, named the RTM worm after Robert T. Morris, its author, entered the annals of information security as the first worm. The RTM worm propagated by exploiting vulnerabilities that let it compromise remote Unix systems. After gaining access to a new system, it compiled and ran a new instance of itself on that system and then attempted to infect of all neighboring systems.

Although the RTM worm compromised less than 5 percent of the reported 60,000 computers interconnected through the Internet at the time, it caused serious availability problems owing to the excessive network traffic it generated and the lack of needed controls to reduce its propagation rate. Within hours of its inception at Cornell University, the RTM worm became an Internet-wide problem that sent network administrators and security officers into a frenzy for days. (In comparison, the Code Red worm, which appeared on 19 September 2001, infected approximately 350,000 computers in 24 hours. See www.caida.org/analysis/security/code-red.)

Since then, several other worms have appeared in a much more populated Internet, all of which share the original RTM worm's basic exploit-and-propagate characteristics and vary in the sophistication of their exploitation techniques and propagation capabilities.

The appearance of the Ramen worm[1] in late 2000 showed that worms were once again a tangible treat. Early in 2001, the infamous Code Red[2] and Nimda[3] worms spread rapidly across the Internet with a number of infected systems and propagation rate never seen before.

On 13 September 2002, a new worm, known as the Slapper, surfaced in the wild in Romania (see

http://online.securityfocus.com/archive/1/291748/2002-09-07/2002-09-13/2). Like its predecessors, this new threat could propagate without human assistance, contained exploit code for a known vulnerability in a widely used Web server program, targeted a specific operating system, and installed and ran copies of itself on infected systems.

But the Slapper worm also revealed a new degree of sophistication in worm technology: using networked instances of its program in a peer-to-peer topology. Until Slapper appeared, all earlier worms captured in the wild (that is, found on compromised hosts) had primitive means of communicating between *nodes* (infected hosts running the worm program) and intended to do little else than quick propagation and direct damage. Slapper nodes represent an evolutionary step toward worms that deploy multipurpose agents, which until this point had only been discussed in academic circles.[3]

What is Slapper?

We can prove that the Slapper is a variation of the Apache Scalper worm by comparing the source code at http://dammit.lt/apache-worm/apache-worm.c (see the "Slapper's predecessor" sidebar) . Modifications introduced in the Slapper worm improved the robustness and efficiency of its predecessor's simplistic P2P networking capabilities. Slapper's author also removed certain features from the original—either because they were redundant or to reduce the perception that it was a tool developed to cause direct harm to networks.

Among the features the author removed from the Slapper were capabilities to update itself from a remotely specified Web server (perhaps to prevent someone else from replacing this version with a new one), to attack and infect a host specified with a controlling program, and to send spam. Interestingly, the ability to execute distributed denial-of-service attacks on a controlling user's behalf was kept intact. Slapper's author attempted to make communications with a remote controlling program as stealthy and untraceable as possible by removing several commands to query status and obtain feedback from Slapper nodes.

Improvements to Slapper's P2P protocol include support for reliable message delivery to nodes, node synchronization, and message routing and broadcasting using a technique the author calls *segmentation*. Using these new networking capabilities for communication with the remote controlling program made the source not untraceable, but at least harder to pinpoint.

As mentioned earlier, the Slapper worm replaced Apache Scalper's original attack code with exploit code for the OpenSSL vulnerability, which was targeted against a combination of at least six different Linux distributions and nine different minor versions of the Apache Web server program.[4] This made the target space for infection considerably bigger than its predecessor. Slapper would attempt to remotely compromise systems by randomly selecting a network to scan and doing a sequential sweep of all IP addresses in the network while looking for vulnerable Web servers.

Slapper's most interesting and innovative features relate to its implementation of a P2P network and the

Slapper's predecessor

On 28 June 2002, the Apache Scalper worm was discovered in the wild, spreading through Internet Web servers running the Apache Web server daemon on FreeBSD operating systems. It used exploit code previously published on security mailing lists to gain access to Apache Web servers susceptible to a recently published security bug, the "Apache-chunked-encoding" vulnerability.

After successfully compromising a vulnerable system, the worm installed and ran itself, turning the compromised system into a node of a peer-to-peer network that connected back to the compromise originator (the parent node) and made a vast array of features available to a hypothetical client program (supposedly under the worm creator's control). The Apache Scalper worm propagated using a simple network scanning mechanism to identify vulnerable hosts.

Features included in the original Apache Scalper worm software included the ability to:

- Launch denial-of-service attack using user datagram protocol (UDP), TCP, and DNS flooding techniques
- Harvest email accounts from the compromised systems' files
- Send multiple email messages (spam)
- Run arbitrary commands on the compromised system
- Attempt to exploit and infect a client's system
- Upgrade the node worm program

Additionally, worm nodes could communicate using a simple P2P networking protocol. Internode communications and communication between nodes and the controlling client were carried over UDP.

The Apache Scalper worm did not have much of an impact on the Internet, perhaps because of the limited installed base of target systems (FreeBSD with unpatched Apache Web servers) or because of its simplistic and inefficient P2P protocol.

A command can be forced to bounce around the P2P network before it reaches its destination mode.

possible implications of this technology in future worms. Let's take a closer look.

The peer-to-peer protocol

The Slapper worm's P2P communications protocol was designed to be used by a hypothetical client to send commands to and receive responses from an infected host (a node). In this way, the client can perform several different actions while hiding its network location and making communications more difficult to monitor.

The P2P protocol is implemented using UDP (at port number 2002) as the transport mechanism. Although UDP is an unreliable transport, the worm's P2P protocol includes a reliability layer on top of UDP. This layer uses acknowledgments and retransmission to build some level of reliability for messages sent in the P2P network from one hop, or node in the worm's P2P network, to the next one.

The worm's P2P protocol stack was written to render it architecture-dependent. Its code does not use host-to-network byte ordering routines, and packet layout is defined by C structures—a form of data representation in the C programming language—that does not use size-specific data types. Because the worm was designed to attack Linux systems running under the I32 architecture, data sent over the network is in *little endian* byte order, while a `char` is 8-bits or an octect wide, a `short` is 16-bits or two octects wide, an `int` is 32-bits or four octects wide, and an `unsigned long` is 32-bits or four octects wide.

Because the author makes no provision for C structure alignment by adding dummy members, the packets have areas that are not accessible via the C structures. For example, the `llheader` structure, when sent over the wire, will include a three-octect unused area.

```
struct llheader {
  char type;
  unsigned long checksum;
  unsigned long id;
};
```

While this structure's members use only 9 bytes of memory, the structure's alignment makes it 12 bytes long when it is sent over the wire.

Reliability layer

The reliability layer adds a header between the UDP layer and higher layers. This header consists of a `signed char` as a single octect that represents whether the packet contains a message or a message acknowledgement (0 for a message, 1 for a message acknowledgement), an `unsigned long` as four octects in the infected system's byte order representing a checksum over the packet's payload, and an `unsigned long` as four octects in the infected system's byte order representing a message ID.

The following is the C structure that defines a packet header

```
struct llheader {
 char type;
   /* 0 = message, 1 =
mes-          sage
acknowledgement */
 unsigned long checksum;
 unsigned long id;
   /* message id */
};
```

When a node sends a message, it keeps a copy in its message queue. The information associated with the messages in the queue includes the message ID (`id`), the time they were first sent (`time`), the time of their last transmission (`ltime`), their destination IP addresses (`destination`), their destination UDP port number (`port`), and the number of nodes to send the message to (`trys`).

In C, the message queue structure looks like

```
struct mqueue {
 char *packet;
 unsigned long len;
 unsigned long id;
 unsigned long time;
 unsigned long ltime;
 unsigned long
    destination;
 unsigned short port;
 unsigned char trys;
 struct mqueue *next;
} *queues=NULL;
```

When sending a new message, the node assigns it a new message ID, which it generates randomly. The node keeps track of the last 128 message IDs it has assigned to outgoing messages and that it has seen in incoming messages. When picking a new message ID for outgoing messages, it will select one that differs from any of the message IDs on this list. An incoming message is ignored if its message ID is on the list of last-seen 128 message IDs. This provides a basic level of protection against receiving and acting on the same message more than once.

The number of nodes to send the message to is only used when the message is to be sent to random nodes in the P2P network, in which case the destination IP address field goes unused and is set to zero, and the UDP port number is the default P2P network port number (UDP/2002). If it sends the message to a specific IP address and to a specific UDP port, the number of nodes to send the message to is set to one.

When the node receives an acknowledgement, it searches the mes-

sage queue for a matching message ID, and reduces the number of nodes to send the message to by one. If the number of nodes to send the message to reaches zero, the node deletes the message from the queue.

At a frequency of every three seconds (the period could be longer), the node goes through the messages in the message queue. If it finds a message that has been in there longer than a certain amount of time, it reduces the number of nodes to send the message to by two. Because the worm never sets the number of nodes to send the message at greater than two, it results in the message being removed from the queue.

The amount of time after which to discard messages in the message queue is a function of the number of connections (`numlinks`) to other P2P network nodes. The function is

```
timeout = 36
          + ( numlinks
          / 15 ) seconds.
```

For a node with 100 links, the timeout would be 42 seconds. For a node with 1,000 links, the timeout would be 102 seconds.

If the message were in the message queue less than the amount of time necessary for it to be discarded and if it were transmitted more than a certain number of seconds ago, then the message would be retransmitted. The minimum period between transmissions is six seconds for messages directed to a specific address or three seconds for messages sent to random nodes in the P2P network.

Command layer

The next layer up in the P2P protocol is the command layer. In this layer, a node can be commanded by a user running a controlling client program can command a node to take some action or respond to commands. The command layer adds a header that includes a command type in the form of a signed `char` represented as a single octect, a command ID in the form of a signed `int` represented as two octects in host byte order (little endian), the length of the command payload in the form of an `unsigned long` represented as four octects, and a sequence number in the form of an `unsigned long` represented by four octects.

In C, the routable layer header looks like

```
struct header {
       char tag;
         /* command type
        */
       int id;
        /* command id
        */
       unsigned long len;
        /* payload
        length       */
       unsigned long seq;
        /* sequence num
         ber          */
};
```

The command ID is a channel identifier; that is, it identifies specific instances of a command. For example, when a client requests the creation of a bounce (a network connection redirection and laundering service), any commands from the client with data for the bounce must use the same command ID as the message that created the `bounce`. Any data generated by the `bounce` sent to the client via a command uses the same command ID as the message that created the `bounce`. The sequence number has a purpose similar to the message id in the reliability layer. It identifies a command instance so that duplicates can be ignored. Its name is misleading given that it is assigned randomly rather than in sequence.

Initialization

When a new node is started (that is, when a computer is infected) it binds and listens to the P2P network's UDP port number, 2002. The new node is passed the IP address of its parent (the infecting node) via the command line. When starting up, the new node will attempt to register with the P2P network by sending its parent a `join network` request command. The parent responds with a `your IP address` command, and a list of known nodes. The parent also broadcasts the new node's existence to the network via a `route` command.

If the new node has an empty list of known nodes , which could indicate a failure to communicate with its parent and to infect new hosts, it will again attempt to join the P2P network by sending its parent a `join network request` command approximately every 60 seconds.

If the new node has a nonempty list of known nodes but has not yet received a `your IP address` command, it will send a `my IP address request` command to two random hosts every 60 seconds.

If the new node fails to register with the P2P network as described earlier, then the P2P network will split. Any new nodes infected by the cut-off node (the node that did not register to the parent's P2P network) will be isolated from the original P2P network. In this way, multiple independent P2P networks could be created.

Routing

The P2P network uses an interesting mechanism to route messages from a source node to a destination node. When a node wants to route a command through the P2P network, it can encapsulate the command in a `route` command, instead of sending it directly to the target node. This command includes the destination node's IP address and a hop count, the minimum number of intermediate nodes between source and destination to pass the message through.

When a node receives a `route` command, it checks whether the destination node's IP address is its

The attackers network traffic does not pass through a single node at any time.

own, and if so, decapsulates the command in the `route` command and forwards it to itself for processing. If the destination node's IP address is not its own, it checks whether the number of hops has reached zero or is greater than 16, in which case it will forward the `route` command to the destination node. Otherwise, it decreases the hop count and forwards the node to two random nodes in the P2P network.

Using this mechanism, a command can be forced to bounce around the P2P network before it reaches its destination node. This affords the attacker a level of anonymity because a node that receives commands from the attacker's client software via the P2P network can't simply examine the packet's source IP address to determine the attacker's network location. This also makes it difficult for any single node to monitor the attacker's activity because it selects a random path through the P2P network each time, and therefore the attacker's network traffic does not pass through a single node at any time.

Because this routing algorithm forwards a copy of the `route` command to at least two nodes at each hop, and when the number of hops reaches zero each node with a copy of the message will forward it to the destination node, the destination node receives 2^H copies of the same command, where H is the number of hops. The default five hops the worm uses results in 32 messages. For the maximum 16 hops, the result is 65,536 messages. The destination node attempts to only process the same command once by keeping track of the last 128 sequence numbers it has sent or received. The length of this queue, 128, seems small compared to the large number of commands a node is likely to see given this algorithm and the size of the P2P network, and is likely to result in duplicate commands being processed. The worm's author named this technique *segmentation*.

Broadcasting

The worm uses broadcasting to announce when a new node joins the network. It also synchronizes the list of known nodes between nodes. The routing mechanism described earlier is used to broadcast messages to the P2P network by setting the destination node's IP address to zero in the `route` command. When a node receives a `route` command with the destination node's IP address set to zero with a sequence number not in the recently seen messages list, it decapsulates the command in the `route` command and forwards it to itself for processing. It then forwards the `route` command to two nodes selected at random.

Because of the random nature of the selection of the next hop destination, this mechanism does not guarantee that a broadcast message will reach all nodes in the P2P network. Slapper's author named this technique *broadcast segmentation*.

Return path

When a node replies to a command, it sends the reply to the IP address that sent the command. This might not be the same as the source node's IP address because the command could have been routed through the P2P network. While the `route` command mechanism routes messages to a destination node, the `route message` does not contain the source node's IP address. How do intermediate nodes in the P2P network route a response command back to the source node? By maintaining a return path routing table that kepts track of the command ID and IP address of the sender of the last 128 messages received.

Client response commands share the same command ID as the request commands to which they correspond. When an intermediate node receives one of these, it looks up in the return path routing table, (by the command's ID) what the next hop's IP address is and then forwards it to it. In this way, the response commands are forwarded to the attacker's client node via the path taken by the last command from the attacker's client with the same command ID.

Synchronization

The P2P network nodes constantly maintain a list of known node in the P2P network and go through great effort to have this list synchronized with each other, to ensure that each node knows exactly the same about nodes in the P2P network. When a new node joins the network, a node that receives the request from the new node sends it a list of known nodes. Additionally, approximately every 10 minutes, each node broadcasts an empty or null `route` command to the P2P network. This command includes the number of P2P nodes the node knows about.

When a node receives such a route command, it checks whether the number of P2P nodes in the route command matches the number of P2P nodes it knows about. If the numbers don't match, and it has been more than one minute since it synchronized, it will synchronize with the IP address from which it received the `route` command. This might not be the same as the IP address of the node that originally created the `route` command because it is broadcasted. In this way, synchronization makes use of the

P2P network protocol and the routing facility. The IP address is that of the latest hop.

When a node synchronizes, it checks whether the number of nodes in the route command is greater than the number of nodes it knows about. If so, it sends a `node list request` to the node from which it received the `route` command (the latest hop). If not, it sends its list of known nodes to the node from which it received the `route` command via one or more `node list` commands. After this, it sets the time of the last synchronization to the current time. It also sets the number of known nodes in the `route` command to the number of P2P nodes it knows about. Finally, it continues to broadcast the message by forwarding it to two random nodes.

Nodes are never removed from a node's list of known nodes. The list will continue to grow even when infected nodes are disinfected and no longer participate in the P2P network. This also means that hosts that were infected once will continue to receive packets from the P2P network even after they are disinfected, until most nodes in the P2P network are shutdown.

Currently, the number of systems compromised by Slapper, its effects, and its networking capabilities are not clear. It is also unknown if anyone has used nodes on compromised systems to launch a coordinated attack or execute specific commands using a remote controlling program. News coverage from as early as 16 September 2002 reported the number of compromised systems ranging from 6,000 to 16,000 and, as administrators patched their systems, a quickly decaying propagation rate. [5,6]

Perhaps because the worm's net outcome doesn't equal the devastating effects of worms such as Code Red and Nimda, the information security community has largely ignored Slapper's sophistication and the glimpse it has provided into the possible evolution of future worms. The subsequent appearance of variations of Slapper (Slapper.B, also known as Cinik and Slapper.C, also known as Unlock) reinforces the assumption that exploit code, viruses, and worms are becoming increasingly complex and sophisticated.[5] They will pose a serious challenge to achieving effective information security if we don't adopt proactive strategy and put substantial efforts not only into fixing newly discovered vulnerabilities quickly but also in trying to prevent their creation. If new threats are not seriously analyzed and their core functionality deactivated—instead of focusing on fixing holes that are only symptoms of the current state of our security posture—we will face far more serious problems in the future.

The conclusion of a recently published paper with a retrospective account of the security evaluation of the Multics operating system almost 30 years ago states this clearly:

> *"In our opinion this is an unstable state of affairs. It is unthinkable that another thirty years will go by without one of two occurrences: either there will be horrific cyber disasters that will deprive society of much of the value computers can provide, or the available technology will be delivered, and hopefully enhanced, in products that provide effective security. We sincerely hope it will be the latter."* [7]

Although our conclusion might not be as apocalyptic as this, if we don't learn from our past experiences and adapt to new situations, we will be purposely downgrading our security intelligence. □

References

1. E.H. Spafford, "The Internet Worm Incident," 1991; www.cerias.purdue.edu/homes/spaf/techreps/933.pdf.
2. E.H. Spafford, "The Internet Worm Program: An Analysis," 1988; www.cerias.purdue.edu/homes/spaf/tech-reps/823.pdf.
3. J. Nazario et al., "The Future of Internet Worms," Crimelabs Research, July 2001; www.crimelabs.net/docs/worm.html.
4. Cert Vulnerability Note VU# 102795, "OpenSSL Servers Contain a Buffer Overflow During the SSL2 Handshake Process;" www.kb.cert.org/vuls/id/102795.
5. M. Broersma, "Slapper Worm Takes on New Forms," http://zdnet.com.com/2100-1105-959385.html.
6. P. Roberts, "Security Experts Divided on Slapper's Threat," www.infoworld.com/articles/hn/xml/02/09/16/020916hnslapthreat.xml?s=IDGNS.
7. P.A. Karger and R.R. Schell, "Thirty Years Later: Lessons from the Multics Security Evaluation," IBM Research Report RC 22534 (W0207-134), July 2002; http://domino.watson.ibm.com/library/cyberdig.nsf/papers/FDEFBEBC9DD3E35485256C2C004B0F0D/$File/RC22534Rev1full.pdf.

Ivan Arce *is chief technology officer and cofounder of Core Security Technologies, an information security company based in Boston. Previously, he worked as vice president of research and development for a computer telephony integration company and as information security consultant and software developer for various government agencies and financial and telecommunications companies. Contact him at ivan.arce@corest.com.*

Elias Levy *is an architect with Symantec. Previously, he was the chief technology officer and cofounder of SecurityFocus and the moderator of Bugtraq, a vulnerability disclosure mailing list. His research interests include buffer overflows and networking protocol vulnerabilities. He is also a frequent commentator on computer security issues and participates as a technical advisor to a number of security related companies. Contact him at aleph1@securityfocus.com.*