Best Defense

NETWORK-BASED PROCESS TABLE ATTACKS

By Simson L. Garfinkel

The attack reached its peak at approximately 4:30pm. I was sitting in my office at Vineyard.NET, an Internet service provider on Martha's Vineyard, typing at a shell window connected to my ISP's primary Web and mail server. Suddenly, the computer printed something on my screen that was tremendously disturbing. I had asked the computer to list the files in the current directory. The computer had told me that it was unable to do so:

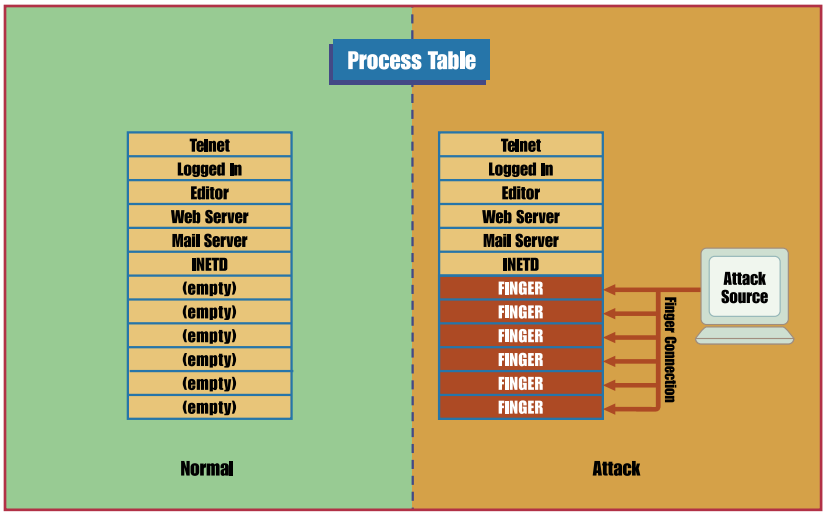
```
% ls
No more processes
%
```

This error message is printed by the C-shell. It means that the UNIX operating system has so many processes running that it cannot create a new process for the `ls` command to execute. This is profoundly bad: practically everything on UNIX — from starting up new login sessions, to receiving mail, to the hourly maintenance programs — requires that a new process be created. I hadn't seen this particular error message in nearly ten years. Back in 1988, when I started writing my book *Practical UNIX Security* (co-authored with Purdue University professor Gene Spafford), I mounted a bunch of attacks against my UNIX system. We called one a process overload attack — a simple programmed attack in which one program repeatedly spawns off new processes. The program that executes the attack is just five lines long:

```
main()
{
 while(1)
  fork();
}
```

Process table attacks can be deadly to a UNIX system. The attack saps the computer's CPU power by constantly creating more processes, each of which is, itself, trying to create more processes. At some point, all of the slots in the process table are filled up, and no more processes can be created. Process table attacks can be very difficult to fight while they are in progress. That's because in order to kill a process, you need to know its process ID, and the only way to find a process ID is by using the `ps` command — which itself requires the ability to spawn off yet another process. Instead, the standard procedure is to reboot the computer and find the attacker.

But the attack on Vineyard.NET didn't look like the attacks that I had launched against myself back in 1988. For starters, the main Web server had plenty of CPU cycles to spare. It just didn't have any processes left. Something else was going on.



Putting A Finger On It: The attack generates so many processes that eventually all the slots are filled up and no more processes can be created.

I typed `ls` a few more times; one out of four times the command actually executed properly. New processes are created and destroyed all the time on a UNIX system. What was happening, I realized, was that every few times I typed the `ls` command I was fortunate enough to have my `ls` process start up before the attacker grabbed the slot that was just freed.

I decided against rebooting. Though that would solve the problem in the

short run, it wouldn't tell me the source of the attack, or how it was being mounted. No, I needed to figure out where the attacker was coming from, how the attack was being launched. Only after learning this could I then patch the hole.

I used the `ps` command to see what processes the computer was actually running. The processes created by the attack were obvious: our Web server was running hundreds of copies of our finger daemon. And the timing was interesting too. Each process had been started exactly 10 minutes after the previous one. Closer examination of our finger daemon revealed that the program had no time-out — the attacker was simply opening up connections to port 79 of our Web server and then letting those connections sit. Each connection created a new process. Indeed, the attack had been going on all day; we had just noticed it when our process table had been exhausted.

Next I used the `netstat` command to see where the attack was coming from. All of the TCP/IP connections to port 79 were coming from the same IP address. As it turned out, the IP address was from one of our own users. We checked with our terminal server and found the name of the account that was being used. For the purpose of this article, we'll call the account "Fred".

I called my partner Eric Bates, the co-owner of Vineyard.NET, into my office. Eric was immediately suspicious. Eric knew Fred — Fred was a novice user who had no interest in attacks. Perhaps Fred's account had been stolen by someone else. Or perhaps it was Fred's teenage son. So we tried calling Fred, but the phone was busy. (Fred, like most people on Martha's Vineyard, has just one phone line.) With the problem diagnosed, we decided to remediate: we killed all of the finger processes and disabled the finger daemon until we could find the source of the problem. Finally, we killed Fred's dialup connection and telephoned his home.

Suffice it to say, Fred was confused when we called him. He didn't realize that his connection to the Internet had been blown away. He didn't understand how we could be calling him while he was online. No, he said, his son hadn't been using the Internet. Instead, he had been trying to use some Web site's chat service — and the chat service had been giving him problems.

We never were able to figure out how Fred had managed to attack our system and bring it to its virtual knees. Perhaps a piece of buggy chat-system shareware was opening up connections. Or perhaps something about Fred's computer was just batty. In any event, we couldn't ask Fred not to go to that site anymore. We had to fix the problem ourselves.

The fix for finger was quite simple: we added a 10-second time-out to the finger daemon. This immediately solved the problem of the buggy chat software — with the 10-second time-out, we would be terminating the finger connections much faster than Fred's buggy computer was initiating them to us. A better solution, we knew, would be to modify the `inetd` daemon so that it wouldn't spawn off new processes if the system was close to some fixed limit.

A few days later, we noticed that a whole bunch of services, including logging, had stopped working on our main Web server. Apparently these processes had died when they themselves were unable to create child processes. Because we had never rebooted our main computer after the attack, the processes that had died during the attack had never been restarted.

The next day, I did a careful inventory of all of our network services and found that many of them were susceptible to the same attack as `fingerd`. Indeed, the majority of network daemons will set up TCP/IP connections and create new processes, no matter how many processes are already running on the system. Most of these programs either don't have any time-out at all, or else they time-out after a relatively long period of time. An attacker could exploit this by creating new network connections and processes at a faster rate than the system would time-out the old ones. Among the vulnerable network servers were `ftpd`, `telnetd`, `rshd`, `rlogind`, `rexecd`, `uucpd`, `nntpd`, `fingerd`, `identd` and even `sshd`. This presented us with a quandary. On the one hand, we wanted to publicize the attack, because `inetd` and other network programs needed to be defended against it. At the same time, the attack was hugely significant: every version of UNIX, and practically every network service, was vulnerable.

Instead of posting the attack to a mailing list such as BUGTRAQ, I decided to follow a more 'responsible' route.

Instead of posting the attack to a mailing list such as BUGTRAQ, I decided to follow a more "responsible" route. I called up my co-author Gene Spafford at Purdue University and told him about the attack. Spaf told me that he would have a student replicate it. Then, if he thought that the attack was serious, he would notify FIRST, the Forum of Incident Response and Security Teams (http://www.first.org). The FIRST mailing list is monitored by both people who deal with security problems and with the manufactures of operating systems.

Spaf actually put two students to work on the task: Tom Daniels and Diego Zamboni. At first, they had problems replicating the attack. The obvious way to replicate it is to write a single program which creates hundreds of connections to the target machine. This approach doesn't work, though, because the UNIX kernel limits the number of TCP/IP connections that a single process can create. To get

around that problem, I wrote an "exploit" script which circumvented this limit — the script forked every sixty seconds. The child processes would spin off a hundred connections to the target machine and wait. The parent process would keep spinning off new children. After 20 minutes, more than 2000 TCP/IP connections were open between the two machines, and the target machine was helpless.

As it turned out, Spaf's students had figured out the problem and had written an attack program similar to mine. Once it was working, the students discovered another, related security problem — the attacking system often crashed as well. It turns out that the Berkeley TCP/IP stack doesn't handle thousands of simultaneous connections between two systems, especially when one of the systems suddenly crashes.

A week after I told Spaf about the problem, he sent out an encrypted message to the FIRST mailing list. (The mailing list is encrypted because of the sensitivity of its messages.) I sent e-mail to a UNIX vendor that I work with from time-to-time. Then Spaf and I moved on to other projects.

As the months passed, I kept thinking about the process table attack I had discovered. I had seen a lot of other people get credit and publicity for discovering attacks. I kept thinking that perhaps I should publicize this attack and grab some credit for myself. But I never found a good time.

In February 1999, I had an unrelated security problem with a version of UNIX I was using. It was a severe problem, and it happened with a new release of the vendor's operating system. As it turned out, it was the same vendor that I had alerted to the process table attack the year before. Since the new release had just come out, I checked it to see if it was still vulnerable. It was.

One of the most frustrating things about being a computer security professional is watching the behavior of software vendors. Time and time again companies ship products with known security problems or fail to fix problems when they are brought to the company's attention. Often, this frustration is made worse by the proprietary nature of most UNIX operating systems — because the source-code is secret, even if a customer knows that there is a security flaw, he is helpless to fix it (or hire somebody else to fix it).

I expected that there would be a fairly loud outcry following the publication of the [process table] flaw.

One of the reasons that FreeBSD, Linux, and other Open Source operating systems are popular among security professionals is that these operating systems are free from the slackness of the vendors. If you know about a security problem, you can fix it. Hand-in-hand with the growth of free operating systems has been the growth of "full-disclosure" mailing lists, such as BUGTRAQ. People use these lists to publicize security flaws — no matter what the consequences — under the theory that it is better to discuss a flaw in public than to send it to the vendors and let them forget about it.

In February 1999, I decided to break my silence on the process table attack. I took the letter that I had written for FIRST and sent it to the RISKS Digest, the ACM forum on computer related Risks that is moderated by Peter Neumann. At the end of the posting, Peter wrote an editorial note saying that I had told him of the vulnerability nearly a year before, that we had notified the vendors, and they had done nothing. A journalist who follows the RISKS digest saw the posting and wrote an article about it for *Inter@ctive Magazine.*

I expected that there would be a fairly loud outcry following the publication of the flaw. I thought that some people would chastise me for making it public, before the vendors had a chance to fix the problem. Instead, I got e-mail from roughly a dozen people. Half of them said that there was nothing new with a process table attack — this sort of attack had been known for years. The other half said that the `inetd` program already had a defense against this sort of attack — the optional argument "max" to the wait/nowait field.

Now it's true, process table attacks have been known for years. But the attack that I had discovered was different because it was network-based. In hindsight, I should have called my attack a "Network-based Process Table Attack", and not simply a "process table attack".

On the other hand, the people who said that the "max" argument in the wait/nowait field defended against this attack either didn't understand the attack, or the meaning of the "max" argument. "Max" specifies the maximum number of processes that will be spawned *per minute* for a particular service. Max has nothing to do with the maximum number of processes that can be running at any given time.

Not much has happened since February. Most UNIX (and Linux) computers on the Internet are still vulnerable to the process table attack. The attack can be mounted either directly from a machine that has been already broken into, or it can be mounted using IP-spoofing. The UNIX vendors have been told and don't seem to care. And many people in the Linux and the Open Source community don't think that the problem is real.

It's all very discouraging.

Simson L. Garfinkel is the founder of Vineyard.NET, an Internet service provider on Martha's Vineyard. He is also a technology columnist for The Boston Globe *and the author or co-author of several books on computer security. He can be reached at simsong@vineyard.net.*