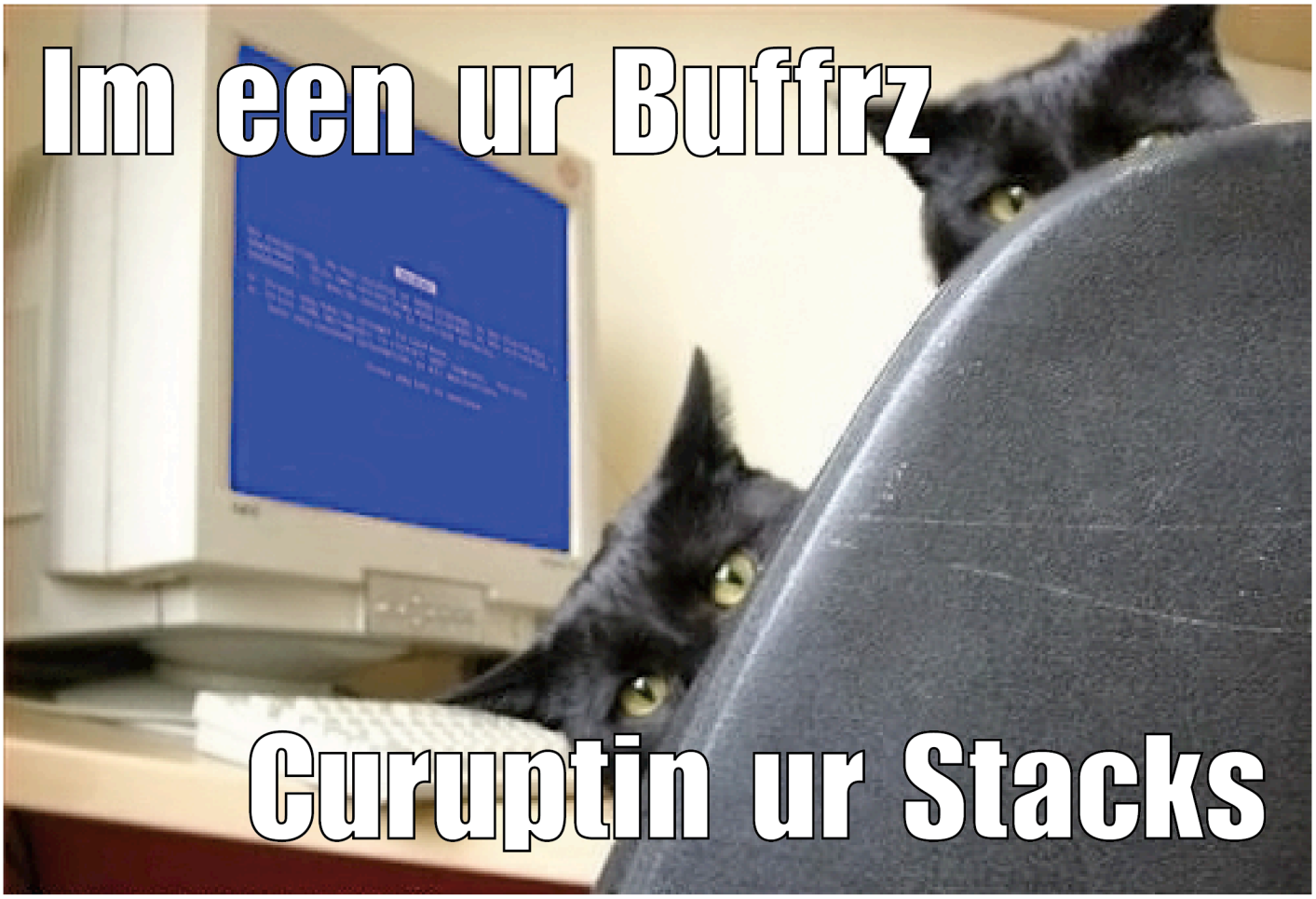
THOMAS CHOPITEA, ADRIEN GSELL
ROOt Lab - BUFFA OVERFLO



STACK-BASED BUFFER OVERFLOWS

The vulnerability we have exploited in this lab was a classic stack-based buffer overflow. Although it is a pretty effective technique to execute arbitrary code on a local or remote machine, the first documented use of buffer overflows dates back from 1988. It was the technique used by the Morris worm to propagate through the Internet back then. The same kind of flaw was exploited by the creators of the Code Red worm in 2001 (it exploited a buffer overflow in Microsoft's IIS), the Slammer worm in 2003 (targeting MS SQL Server) and even in gaming consoles like the XBox or the Nintendo Wii!

That being said, lots of people have since then investigated ways to mitigate this kind of attacks. What follows is a recap' of the most popular techniques to detect and / or prevent buffer overflows in modern computers.

HOW TO FIX ADDHOSTALIAS

Use safe libraries

The most straightforward way to secure addhostalias would be to use safe C libraries in order to manipulate buffers. sprintf is extremely unsafe as it does not do bound checking when writing to the destination buffer, and only expects the '\0' character indicating the end of the data to write. A more secure function should be used, such as strlcpy, which compels the developer to input the amount of bytes to be copied. This means that the buffer will not be written to more than the number of bytes specified, therefore preventing sensitive parts of the stack from being overwritten.

Use safer programming languages

We could also re-rewite the whole application using a language that does automatic bound checking - and is therefore impervious to stack-based buffer-overflows - like Java or .Net (the latter being more unlikely to happen since we are running a UNIX-based OS).

OTHER COUNTERMEASURES

Canary values

A widely used countermeasure to buffer overflows is the use of canary values. Thes values are variables that will inevitably be changed when a buffer in a local variable is overflowed enough to get to the stack.

Local variables are rearranged in the stack so that the buffers are the closest to the new frame pointer. The canary value is then placed between the new frame pointer and the first buffer. In this way, if someone tries to overflow one of the buffers so as to reach the return address, canary value will inevitably be corrupted in the process. Upon check, appropriate measures can thus be taken.

There are many types of canary values that can be used to prevent stack-based buffer overflows:
Terminator canaries: **based on the assumption that most buffer overflow code ends in null terminators ('\0'), these canaries are combinations of NULL characters, carrier returns, or line feeds. The disadvantage is these values can be known, and a skillful attacker could write shellcode comprising these values and "overflowing" them exactly on the canary value - rendering it useless.**
Random canaries: **these have the advantage that they cannot be predicted, and it has to be read by the attacker for him to overwrite the canary value with its initial value - which increases the difficulty of suck attack attempt**
Random XOR canaries: **these are random canaries XORed with other data inherent to the program. This means that if the canary value OR the control data are altered in any way by a buffer overflow, it will be detected immediately by the monitoring system. The attacker has to be extra careful when manipulating these, and it renders the attack even more difficult.**

Executable space protection

This technique consists in preventing code from the stack to be executed, even if an attacker has managed to place executable code into a buffer on the stack. However, redirection to other program functions can still be used.
Thomas Chopitea (880919-T276) / Adrien Gsell (880528-T253)

Network layer

The network layer can also offer a certain amount of protection against buffer overflows. Buffer overflows usually share the same structure, a large NOP slide followed by some kind of shellcode (usually trying to spawn a shell, /bin/bash for example). This signature can easily be integrated in Deep-Packet Inspection firewalls or Intrusion Detection Systems and raise alarms or block certain packets that match the heuristics.

Although this is an effective protection mechanism against remote buffer-overflows, it does not protect the system if the attack comes from the local host (privilege escalation attacks, for example). In addition to that, dealing with application security at the network level is usually a bad idea. For example, if a firewall starts to malfunction, or packets take a different, unprotected route, protection will be completely bypassed. If the code itself is secure, then the application's security will be totally independent from the network architecture.