Testing the SETI-Hacker Hypothesis

Marcus Leech, VE3MDL

Seti League Observer

Abstract

In a recent paper[1], Richard Carrigan forms a hypothesis on the existence of a malevolent ET signal, intended to infect another intelligent civilizations computers with a destructive virus. In this paper, we test some of the elements of this hypothesis and find them to be unsupportable.

Introduction

In a PhD thesis by Leigh[2], it is shown quite convincingly that one-way digital communications from an ET civilization within a few dozen light-years from earth is entirely within the realm of the physically possible. Leigh uses a variant of the well-known *Shannon Law* to show that such communications paths are entirely possible, given a suitably powerful transmitter, prudent selection of wavelength, and receiving technology no more advanced than current-day technology in use by radio astronomers here on earth.

A later paper by Carrigan posits the existence of malevolent signals that could constitute a computer virus, and that SETI researchers should take special care, assuming that they ever find a suitable signal, not to allow the resulting “bits” to be executed as code. Carrigan recommends a so-called quarantine for SETI data, in order to prevent a theorized globally catastrophic outbreak of computer malaise.

[1] See: http://home.fnal.gov/~carrigan/SETI/SETI%20Hacker_AC-03-IAA-8-3-06.doc

[2] See: http://seti.harvard.edu/grad/dpdf/thesislt.pdf

The communications channel

Carrigans brief paper assumes a channel of approximately 100kbit/sec capacity, based on reasonable assumptions about antennae sizes, transmit power, receiver noise figure, etc. None of the assertions about the *Shannon Law* capacity of the hypothetical communications channel can reasonably be disputed.

For purposes of this paper, we'll assume a 100kbit/sec channel from our purported ET adversary. We'll also assume that they have correctly surmised that target civilizations have converged on the use of binary coding for machine logic systems. That humans happened to pick binary was not entirely a forgone conclusion at the start of the 20th century, and it would be wise to remember that other systems of machine logic briefly occupied scientific and engineering imagination—trinary and decimal were not unheard of, as well as other multi-level logic systems. We also assume that our ET has an Arecibo-sized antenna, with high efficiency, and that we use an Arecibo-sized antenna, or an array with equivalent collecting area and aperture efficiency.

So, for purposes of further dialog, we'll assume that our ET knows its victim is going to use binary.

We also assume that once our hapless victims (the SETI astronomers) have found such a thunderingly-huge signal, that they'll continue to track it precisely, and record every single bit of the demodulated data. Under the assumptions put forth by Carrigan, a wavelength of 3 cm is chosen for the communications channel, which means that a beam width of only 0.007 degrees (for Arecibo-sized antennas on each end) must be maintained accurately to track the signal. Our malevolent ET must also cause their antenna to track the tiny beam at a particular “target” for long enough to have some reasonable effect. We assume that they've already identified us by our TV and radio transmissions as a likely target.

ET hacker assumptions

Overall in this paper, we'll be making assumptions that are as generous as possible in favour of our ET hacker. We can assume that they're at least as smart as we are, but probably not orders-of-magnitude smarter, since if they were, they wouldn't bother attacking a neighbouring star systems computer networks. They'd probably find other things to amuse themselves.

We'll also assume that it isn't a problem that there are a myriad of possible computer architectures “in play” at any given time in human history, many of them significantly different from one another.

Since our nefarious ET friends theoretically have no way of conducting a black-box analysis of their target (since the communications path isn't two-way in any practical sense), they cannot apply the usual black-box techniques for determining how the target system works.

We've already said that they must have assumed a binary system for machine logic. There are a very few other assumptions that our wicked ET could make.

Human computing activity in the last 50 years has nearly-universally settled on a collection of bits, called a byte, to represent data. Many types of data are encoded into these “bytes”. But the byte emerged not out of natural, overwhelming necessity, but almost entirely by cultural accident. The English language, having 26 characters in it, can be represented handily in both upper and lower case forms using roughly 6 bits. Add in some punctuation, and a few control characters, and the nearly-arbitrary boundary at eight bits emerges. It could easily have emerged that we'd settled on 7, 9, 10, or 11 bits as “natural quantum”. References to computers, with superficial representations of their architecture didn't start to appear in popular human culture to any great degree until the 1980s. We can assume that our smart ET criminals will have used the subtle clues in our television and radio transmissions to intuit that we use a natural bit-collection of eight bits, and reasonable multiples thereof. It's a bit of a stretch, but we try to put the Carrigan hypothesis in the best possible light in order to proceed.

Our ET hackers could assume that buffer overruns were a very common flaw in computer programming, since presumably their own civilization has been through it themselves. Which means that they have to decide how big our buffers are, and how to best make use of overflow.

For example, here on earth, buffer sizes happen to be picked based on multiples of 512, which is a power of two. Our ET can assume that data buffers will also be sized to be a power of 2, since we use a binary logic system. Using that logic, and assuming that the victim has technology within a couple of hundred years of the attackers, then buffer sizes could range between 128 bytes and 65536 bytes (1024 bits and 524288 bits), if one also assumes buffer sizes that are exact powers of 2, that's only a handful of different buffer sizes to try, in order to execute an overflow attack against our unwitting SETI researchers.

Our ET villains need to make other assumptions about our technology as well. In order for a buffer overrun attack to be fairly effective, the buffer overrun needs to, with very high probability, immediately affect the flow-of-control of a running program. The currently-dominant computer architectures here on Earth have evolved an architectural curiosity called the “stack”. The “stack” is an area in memory that is usually used to contain transient variables such as local variables within a subroutine, saved copies of the control and data registers of the CPU, and the so-called “return address”--the address that program flow is to return to when the subroutine finishes. There are a variety of possible architectures that don't use the “stack” concept, and several extant in current human endeavor, but let's assume that our ET has at least heard of the concept, and understands the implications for malicious code.

The stack on a subroutine call typically looks like this:

```
return address
saved register 1
saved register 2
saved register 3
...
saved register N
local variables
```

Together, that collection of “stuff” on the stack is called a stack frame. The details vary quite a bit from CPU architecture to CPU architecture, but the concepts remain the same. It's important to note that in general the stack grows *downwards*, while the regular program data (the heap) grows *upwards*. It's generally the case that the stack starts at a very high (virtual) address, while the instructions and heap data start at a very low (virtual) address. The “goal” of a buffer-overrun targeted at the stack is to precisely replace the return address with an address that points to within the memory occupied by the buffer we just overran. This scheme works very well for overrunning buffers that are declared as local variables (and thus allocated on the stack). A buffer of 512 bytes, for example, will be allocated 512 bytes on the stack, with indexes into the buffer growing *upward*, towards the return address and saved registers. Hackers here on earth know this, and take ruthless advantage of it, but only against *programs that are known in detail in advance*. The precise layout of the stack is dependent on the language compiler that produced the program, and the architecture the program is running on. Further, the precise relative locations of local variables, and the buffer we wish to attack is **exceedingly important** in mounting a stack-smashing buffer overrun attack. Hackers here on earth get this information either through examining the relevant *source code*, or examining and reverse engineering the *machine-code* binary executable rendition of the victim programs. There has been some small amount of research into so-called *blind stack smashing* attacks[3] here on earth, but such attacks require two-way feedback—the so-called *black box* approach. Our ET vandal doesn't have the luxury of seeing the source code, or even getting to reverse-engineer the binary executable code.

They must necessarily use a random approach to the attack problem. They are further hampered by being entirely unaware of the predominant *source language* of our computer programs, nor are they aware of the *idiomatic constructions* emitted by our compilers when translating source code to machine code. It is precisely these idioms that hackers take advantage of when attacking actual programs (that, and human programmer frailty—properly-constructed programs cannot be attacked through their data inputs).

We need, then, to establish the magnitude of the problem our ET miscreants face when trying to blindly, randomly, attack the SETI researchers computers (and by computer viral propagation, the rest of the computers on the planet).

The stack frame problem

Before one can smash a stack, one needs to have some estimates as to its layout. The previously shown “typical” layout for a stack frame is, of course, entirely useless in practice. In practice, there are an unknown number of CPU registers, of an unknown size, and virtual addresses are also of an unknown size. We can be extremely generous, and assume that our ET has concluded that we use 32-bit addresses and instruction words, and that our “register file” on the stack is between 8 and 64 registers, of either 32 or 64 bits per register. Somewhat surprisingly, virtual addresses of roughly 32 bits have been a technological constant in computer architecture since the 1960s—while detailed architectures have changed radically in the last 40 years or so, virtual addresses of roughly 32 bits are a very good bet, along

[3] See, for example: http://www.ngssoftware.com/papers/NISR.BlindExploitation.pdf

with CPU integer registers of roughly the same size. Having determined our rough technological advancement level through our TV and radio transmissions, our ET friends might derive a virtual address and register size somewhere in the neighborhood of 32 bits.

The fact that register sizes are roughly 32-bits means that they can assume that the stack starts somewhere around 0xFFFFFFFF, and grows downwards, and that the program and heap data starts somewhere around 0x00000000, and grows upwards. For a typical modern computer program, there may be several 10s of kilobytes on the stack at any point in the life of an executing program—possibly an order of magnitude more if most subroutines allocate buffers from the stack. Let's assume that our ET knows that stack addresses start at 0xFFFFFFFF and grow downwards to somewhere around 0xFFFD0000. In the real world, however, different operating systems layout the virtual address space differently. On on recent Linux systems, for example, the stack starts somewhere around 0xBFFF0000, but the actual value appears to be different from run to run—perhaps to deliberately interfere with stack-smashing attacks. If you don't know exactly where the stack starts for a given program, it's hard to build a “generic” attack, since you need to know actual virtual addresses.

Here we see a code snippet from a typical C program:

```
int a, b, c;
unsigned char buffer[1024];
double foo;
double bar;

. . .

[the code of the subroutine]

return;
```

The attacker assumes that somewhere in the “code of the subroutine”, the code takes data from an external source, and stuffs it into *buffer*. The assumption is that the code doesn't adequately check for overflow of the buffer, which allows the hacker to overwrite the values for a, b, c, and the rest of the stack frame, including the return address where execution will resume after the subroutine finishes its work. Our ET doesn't know anything about the subroutine, so they can only guess about the relative positioning of a buffer, any local variables that may be higher in memory than the buffer, and the contents and extent of the stack frame. The only thing our ET really knows is that it will be radio-astronomy software that will be receiving and processing their bit stream. This constrains the problem somewhat, but it's still a formidable task. For sake of argument, let's say that there are 1.0e8 possible stack layouts corresponding to radio-astronomy software designed for receiving trivially-modulated SETI signals. Let's say that the stack layout part of our evil payload varies between 256 bytes and 4096 bytes, in 4 byte increments. At 10kbytes/sec, that's between 25milliseconds, and 0.4 seconds to transmit just the part that corresponds to the stack layout. If we assume that only one of the 1.0e8 possible layouts is the actual layout, then it will take between a month and 1.5 years just to send the data bits necessary to “hit” the exact magic stack layout. Keep in mind that's just the amount of data necessary to cause a buffer overrun to hit the return address on a stack whose layout we're entirely, utterly, uncertain of. We haven't gotten to the part where we actually make this magic **do** something, all we've done is overwritten the return address on the stack. But, of course, we don't **know** that we've done that, since we have only a one-way communication between our victim (the hapless SETI scientists and their innocent computer software) and us. Which means that we can't build an incremental approach to the attack—it's all or nothing. Which brings us to the next part of our tough problem.

The instruction set and precise virtual addresses

The first part of the problem that's really tough is to place the correct virtual address in the return address portion of the stack frame we've just accidentally, and with no two-way feedback, smashed. We observed earlier that the stack **concept,** combined with the notion of 32-bit address words implies that the stack may start somewhere around 0xFFFFFFFF, although actual implementations must necessarily change this. Some systems randomize the exact start address of the stack, in order to foil stack-smashing code. The total number of stack frames currently on the stack, and their aggregate size, is unknown to the attacker. But let's make some very generous assumptions, and assume that the total stack-frame aggregate for any program in the class of programs previously described (SETI radio astronomy software) varies over a range between a few hundred bytes, and a few hundred thousand bytes. That leads to virtual addresses that are in the stack that vary by only a factor of a thousand or so. The top of the stack will be somewhere “up there” in most implementations. For sake of argument, let's say there's only about 64 different possible “top of stack” addresses, with some fuzzing of perhaps a few million addresses. Current Linux, for example, seems to fuzz the stack top over a range of roughly 8 million addresses. It would be reasonable to assume an uncertainty in top-of-stack address of perhaps 1.0e7 addresses. Which means that when we modify that return address, we have a total uncertainty of roughly 1.0e9—which is perilously close to the maximum number of addresses that can be represented by a 32-bit integer!

Our attack vector has an uncertainty of roughly 1.0e8 uncertainty in stack layout, multiplied by an uncertainty of 1.0e9 in the virtual address it needs to “paste” into the return address field on the stack frame. Combining these two uncertainties, the probability of any given trial succeeding against our program has a **lower** bound somewhere around 1.0e-17. How long will it take to transmit these still-not-fully-fleshed-out attack vectors? Roughly 80 million years, although, on average, one can expect success in half that time—that's assuming the roughly 10kbytes/sec in the Carrigan hypothesis.

Let's assume that we've successfully pasted the return address with a correct value—a value that lies somewhere in the data-buffer that we control. What to put into that data buffer? Our attacker needs to have something to put in there that will be particularly devastating, hard to detect, and propagates quickly and stealthily. We'll give our attacker the benefit here of running into particularly non-security-savvy SETI scientists, who run their analysis tools with so-called “root” privilege, in order to effect maximum devastation. Here's a lethal little code snippet, that would work on a Linux system:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
main ()
{
        int in, out;
        int i;
        char buffer[1024];

        in = open ("/dev/urandom", O_RDONLY);
        out = open ("/dev/hdXX", O_WRONLY);
        for (;;)
        {
            read (in, buffer, 1024);
            if (write (out, buffer, 1024) != 1024)
            {
                break;
            }
        }
    }
```

This piece of code overwrites a system-disk with random data. It likely won't run to completion before the system crashes, but it serves to show an example of a class of programs I call “minimal lethal payloads”--that is, program payloads that are short, and nearly immediately lethal to the “host”. The above program compiles to roughly 1200 bytes on a modern Linux system, but one can assume that a hand-coded assembler program could compact this number down by perhaps a factor of 3. So, the lethal part of the payload may be around 400 bytes. Our ET doesn't know anything about the instruction set of its victim, and it certainly doesn't know anything about files system layout, the mechanisms behind system calls, etc, etc. Exactly how many 400-byte sequences are there? Really a lot—roughly 1.0e980 of them. That's right—1 with 980 zeros after it. The density of such sequences that are semantically useful programs in the target environment is very low, further, the number that are immediately system-lethal is even lower. Let's make a wild-eyed guess and say that there are maybe 1.0e8 immediately system-lethal programs that are 400 bytes in length. The actual number may be several orders of magnitude lower or higher. What does that do for our numbers—it means that roughly once every 1.0e972 trials, we emit a 400-byte sequence that is immediately system-lethal. To calibrate these numbers, the famous *Slammer* worm payload was 376-bytes in length[4]. Note that the *Slammer* worm had no actual “payload” beyond propagation—the only harm it did was slowing down your computer and network interfaces as it tried to propagate to other vulnerable systems on your network.

But we really don't want a “payload” that is immediately system lethal. Immediate lethality is an extraordinarily bad strategy for any virus, computer or biological. Let's assume that our ET virus also needs a propagation phase. Let's be generous, and assume that a further 400 bytes is required to encode the propagation phase. Which means a total payload of perhaps 800 bytes. The total number of 800-byte sequences

[4] See: http://www.cert.org/advisories/CA-2003-04.html

is truly enormous—roughly 1.0e2000 sequences. Let's assume that there's a large number of possible 800-byte propagate-then-exterminate viruses possible in this sequence. There's possibly as many as 1.0e9 such sequences in that space, which means that roughly once every 1.0e1991 sequences, we'll hit on one at random that has the desirable propagate-then-exterminate properties. At the Carrigan-hypothesis rate under discussion (10kbyte/sec), it would take a very long time indeed to transmit all the necessary sequences. Emitting even a minuscule fraction of such sequences before the heat death of the universe is left as an exercise for the reader.

When we combine the probabilities of successful stack-smashing (roughly 1.0e-17) with the problem of generating a semantically-useful program of roughly 800-bytes, we get an overall probability for a single trial of somewhere around 1.0e-2008.

Propagation phase

Any successful apocalyptic "worm" must necessarily have a propagation phase. While sending a "kiss of death" payload across the light years to cause some hapless SETI researchers computer to crash might seem amusing, to really foul things up, our little ET worm has to actually propagate to other computers on the planet, so that they may also participate in the remote-control planet-domination considered in the Carrigan paper.

This immediately brings us back to the architecture-guessing problem discussed earlier. Our ET has no way of knowing how our planet-spanning digital computer networks actually work. Since the 1960s, there have been a very-large number of such computer network architectures. It is only through a set of extraordinary circumstances that most of the planets computer systems currently speak using the TCP/IP protocol suite.

There were a very large number of equally-viable alternatives that we didn't pick, but were in widespread use until fairly recently. Architectures like X.25, Chaosnet, ARCNet, NetBios, DecNET (Phase 1 through 5), the OSI protocol stack, DataKit, UUCP, NCP, SNA, PUPNet. For a while, it was in-vogue for grad students to invent entirely-new network architectures for local-area-network computing, which surely created thousands more dead-ends. But the total number of possible globe-spanning-network architectures for a civilization that has standardized on binary logic systems for both information transfer and machine logic, s very large indeed.

We have to assume that correctly guessing which of the many-possible network architectures the planet is actually using is at least as hard as guessing the dominant CPU architecture.

Conclusions

The scenario described in the Carrigan paper is clearly based on his limited expertise in computer science and computer architecture—the arguments with respect to link budget for maintaining a high-speed one-way link are beyond reproach, but the arguments involving the computer-science aspects are based on inadequately-informed speculation. It is hoped that this paper fills in the information vacuum left by the Carrigan paper. Even when we make exceedingly generous assumptions about the predictive abilities of our hypothetical ET hacker, we come up against overwhelming odds against the ET. An extinction-class asteroid impact is a vastly more worrisome scenario than computer viruses from ET, by several orders of magnitude.

We conclude, with apologies to the film “Independence Day”, that SETI-hacker scenarios are only plausible within the fanciful confines of Hollywood, and then only when our ET hackers happen to be Macintosh™ savvy.

It is amusing to observe that the SETI hacker “threat model” is one of the very few that is adequately countered by the “security through obscurity” doctrine that is generally held in such low regard by both the author, and the computer security community in general. Indeed, it is precisely this *obscurity* that protects us nearly-unconditionally from the hypothetical threat of blind viral attacks by ET hackers.

While one cannot recommend a cavalier attitude with respect to software quality used by our SETI researchers, it's a near-certainty that computer viruses from outer space will not be one of the threats that need to be defended against. Indeed, it seems likely that the religious and social upheaval that would be caused by the mere existence of an ET signal would have more far-reaching societal consequences than the content of that signal. Indeed, under the assumptions in the paper, the content will be uniformly random rubbish, with a probability approaching unity.

Acknowledgments

The author gratefully acknowledges the thoughtful comments of Dr. Paul Shuch of the SETI League, and Dr. Steven Bellovin, CS Department, Columbia University.