“前端加密与混淆”

2017.04 | @EtherDream

关于我

- 网名：EtherDream
- 属性：[Geeker, Hacker)
- 爱好：前端技术、网络安全，探索黑科技

前端加密有意义吗?

常见回答

- 毫无意义，因为前端是公开的
- 不要自创加密算法
- 直接用 HTTPS 吧
- 各种密码学探讨 ……

事实上 ...

即使不了解密码学，也应知道是 有意义 的，

因为 加密前 和 解密后 的环节，是不受保护的。

HTTPS 只能保护传输层，此外别无用处。

加密环节

- 传输加密（对抗链路破解）
- 数据加密（对抗协议破解）
- **代码加密**（隐藏算法、反调试、防黑盒...）

代码保护

- H5 小游戏（防止被简单的扒走）
- 信息收集 （广告作弊检测等）
- 恶意脚本 （躲避杀毒软件）
- 安全防护 （防止 CC、爬虫、假人等）

最早经历

做一个 网游服务端 防火墙。

- 起因

  攻击者伪造游戏协议，制造大量假玩家。

- 传统防御

  在游戏协议层面，进行策略防御。

突发奇想

分析 私有协议 较麻烦，能否用 公开协议 代替?

- 设想

  用通用的 B/S 架构，防御私有 C/S 服务

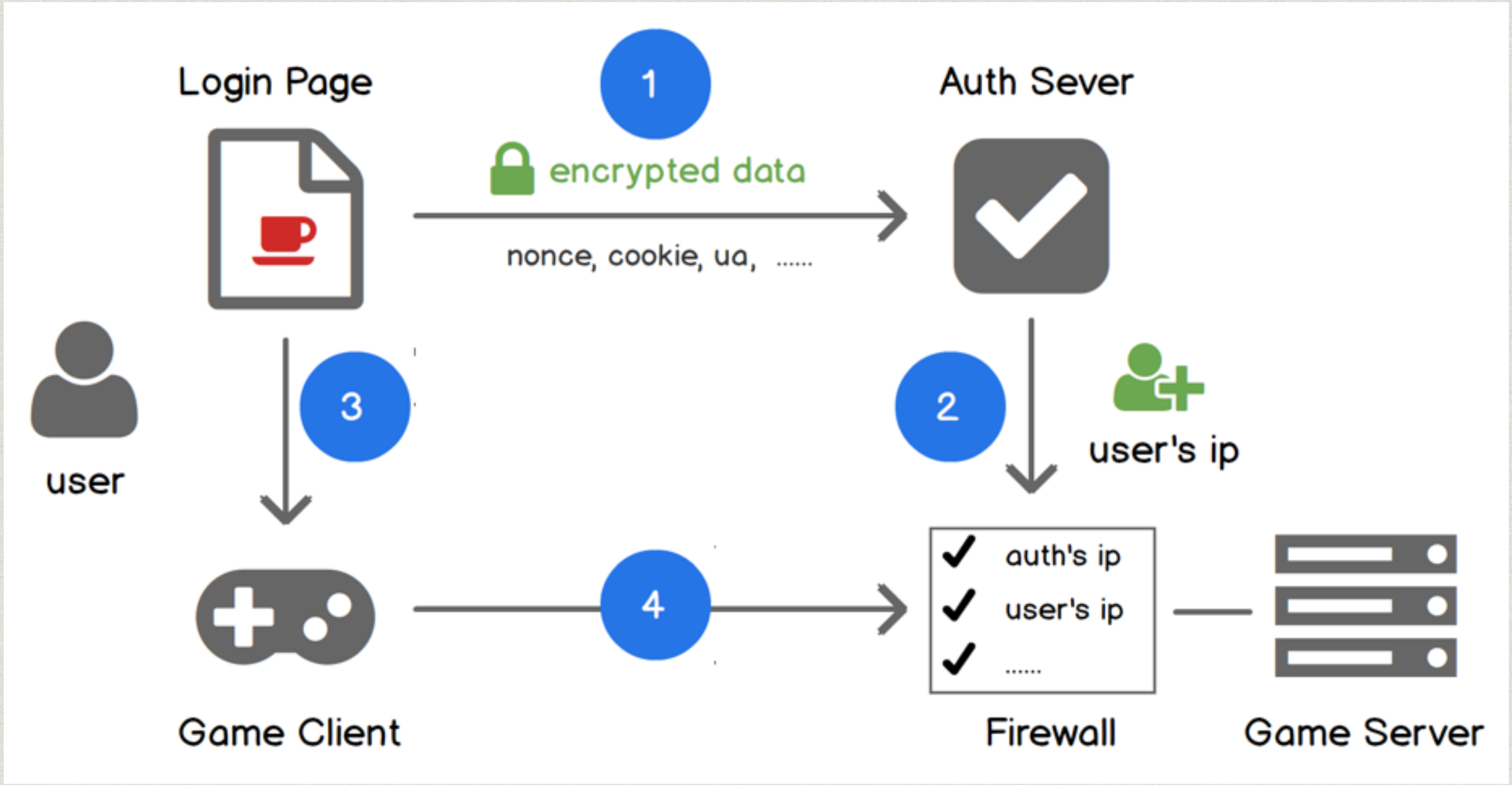
- 优势

  后端可借助 WAF、CDN 等

前端战场

登录器有个 内嵌网页，正好可以部署 JS 脚本！

实现原理


Login Page
1
Auth Sever
encrypted data
nonce, cookie, ua, ...
user
3
2
user's ip
auth's ip
user's ip
.....
4
Game Client
Firewall
Game Server

方案缺陷

前端的一切都是公开的，严重依赖

代码保护

代码保护

- 脚本压缩、加壳
- 检测篡改、黑盒
- 通过 语法树 混淆
- 通过 控制流 混淆
- 通过 虚拟机 混淆

脚本压缩

脚本压缩

去除尽可能多的 有意义信息

- 删除 注释、空格、换行、冗余符号 等
- 变量重命名，变成 a、b、c 等
- 属性重命名，变成 a.a、a.b() 等
- 无用代码移除

脚本压缩

对抗方式

- 通过脚本美化工具，恢复成可读格式
- 分析 全局变量、JS 属性、DOM 属性 等保留名
- 分析 字符串、数字、正则 等常量

小技巧

单例模式，尽可能平坦化

```
util = {
    fnA: function() {
        ……
    },
    fnB: function() {
        ……
    },
}
```

```
function util_fnA() {
    ……
}

function util_fnB() {
    ……
}
```

原因：属性名 难混淆、无用代码 难移除

脚本加壳

脚本加壳

将脚本进行编码，运行时 解码 再 eval 执行

eval (.....

..... !@#$%^&* .....

.....)

对抗：eval 换成 alert 就能原形毕露。

脚本加壳

改进方案：用其他 API 代替 eval

- Function / (function(){}).constructor
- setTimeout、setInterval(“code”)
- <script>code</script>
- <img onerror=“code” … >
- frames[0].eval、opener.eval

脚本加壳

对抗方式：监控 API 调用

- Hook 函数

  ```
  window.setTimeout = …
  ```

- Hook 访问器

  ```
  Function.prototype.__defineGetter__('constructor', …)
  ```

- 监控元素

  ```
  MutationEvent / MutationObserver
  ```

加壳欺骗

99% 的人会尝试 打印 eval 参数，因此可埋设陷阱：

```
~~eval~~ log ( ..........................................................
        ......
        T = setTimeout(bomb, 0)      // 埋一颗定时炸弹
        ......
        code += 'clearTimeout(T)'    // 执行 eval 可解除
        ..........................................................
)
```

如果 eval 未执行，定时器就会触发！

演示：https://codepen.io/anon/pen/RVgaxB

加壳欺骗

定时器触发后。。。

- 日志上报，及时了解情况
- 在本地存储 隐写 特征，长期跟踪
- 释放 CSRF 漏洞，获得破解者的详细信息
- 开启自杀程序（页面崩溃、死循环、耗尽内存等）

加壳干扰

解码时插入无用代码，干扰显示

- 大量换行、注释、字符串等
- 大量特殊字符，导致显示卡顿

对抗方式：不通过 GUI 复制粘贴代码，直接工具过滤

相应攻击：检测运行环境，非浏览器环境下，释放无效的代码 :)

环境监测

代码自检

通过函数 toString 方法，检测代码是否被篡改

```
function Module() {
    ……
    if (Hash(Module + '') != 0x11223344)
        // 代码被篡改!
    ……
}
```

调试自检

检测是否有调试特征

- 控制台是否打开
- 通过算法执行时间，判断其中 debugger 指令是否执行
- Worker 之间心跳检测…

环境自检

检测环境是否异常，识别 黑盒攻击

- 页面 URL 在白名单外（脚本被扒到其他地方运行）

- 用户行为存在异常（数量特别少，或者不合常理）

- 在特殊浏览器上运行（WebDriver、PhantomJS 等）

  如果是 NodeJS 环境，甚至可以启动木马，长期跟踪 :)

延时反馈

检测到异常，不立即反馈，防止过早暴露

```
x = 1

if 检测到异常
    ~~exit()~~                  # 容易暴露，强烈不推荐
    x = 5                       # 先悄悄记下

for i = 0 to N ^ x              # 增加复杂度，让性能变差，但不影响逻辑
    malloc(M ^ x)

val = arr[index % x]            # 甚至可以影响逻辑，偶尔算出错误结果~
```

延时反馈

没有破解不了的代码，能延长时间就足矣

启示：

- 传统软件检测到自身被破解时，经常会有提示，
  破解者往往通过这些提示，反向寻找检测逻辑，从而彻底破解。

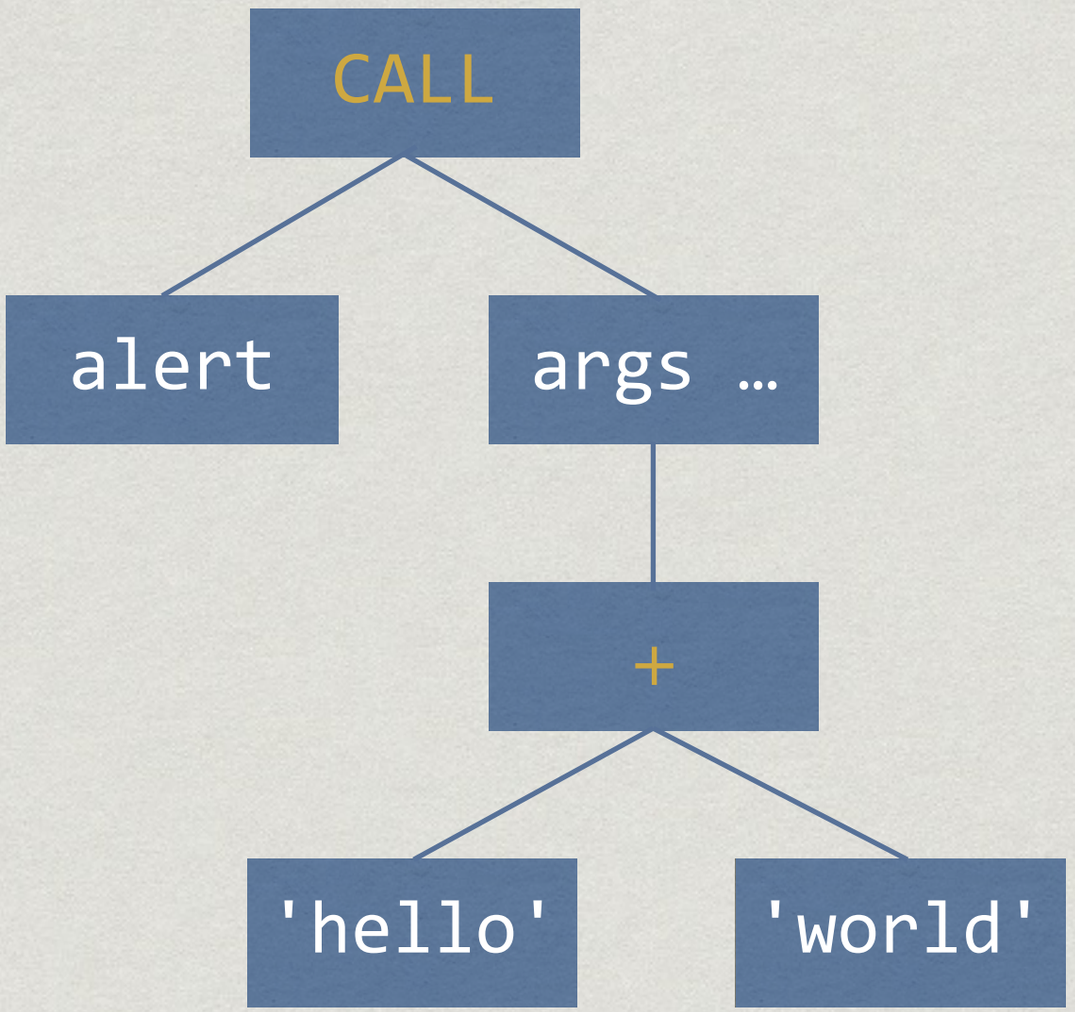
- 更好的思路则不提示，而是间接影响某些参数。
  一段时间后，算法变得更慢更耗内存，偶尔甚至发生一些错误。
  让用户觉得盗版软件问题特别多 :)

语法树混淆

语法树

AST（Abstract Syntax Tree）

```
alert('hello' + 'world')
```



常用工具： UgliyJS、Esprima、Acorn …

常量混淆

通过 AST，重命名常量

```
(function(a, b, c, d, e, f, g, ……) {

    if (a[b] === c) {

        d[e][f](g, ……)

        ……
    }

})(window, 'ver', 123, document, 'body', true, ……)
```

常量混淆

通过 AST 调整字面常量

```
var info = { name: 'jack', age: 20 };
```

```
var t = {};
var info = (
    t['name'] = 'jack',
    t['age'] = 20,
    t
);
```

常量加密

通过 AST 加密常量，运行时解密

```
function DEC(x) { …… return str; }
```

```
if (this[ DEC('#$%^&') ]() === DEC('!@#$') ) {
    ……
}
```

当然，破解也是非常容易的：

遍历 AST，对名字为 DEC 的函数节点 求值，用常量节点替换之

求值陷阱

1. 在解密函数中，埋一个隐蔽的陷阱：

```
function DEC(x) {
  ……
  x == '^-^' && run_evil_code
  ……
  return str
}
```

2. 在一个 永不到达 且 无法静态分析 的分支里，引用该函数：

```
if (xhr.status == 666) this[ DEC('^-^') ]()
```

求值陷阱

正常用户不会执行，而 AST 遍历求值时，则会触发陷阱！

对抗策略

- 避免将复杂的 解码函数 放在自己的脚本里执行

  （如果是 NodeJS 环境，陷阱里的恶意代码或许可删光你的文件~）

- 使用沙盒环境执行 解码函数

求值陷阱

提问

语法变丑

通过 AST，破坏原始语法

```
for (i = 0; i < n; i++) {
    ……
}
```

```
i = 0; do {
    if (i >= n) break
    ……
    i++
} while (true)
```

运算混淆

通过 AST，重载运算符

```
c = x + y - z
```

```
function ADD(a, b) { return a + b }

function SUB(a, b) { return a - b }

c = SUB(ADD(x, y), z)
```

运算混淆

所有运算符，都能变成函数

```
function EQU(a, b)  { return a == b }

function MOD(a, b)  { return a %  b }

function XOR(a, b)  { return a ^  b }

function HAS(a, b)  { return a in b }

function GET(a, b)  { return a[b]   }
```

……

动态执行

运行时动态决定运算符，干扰静态分析

```
c = 1 + 2
```

```
function ADD(a, b) { return a + b }

function SUB(a, b) { return a - b }

f = dynamic_func() ? ADD : SUB;

c = f(1, 2)
```

语法展开

将语法展开，每次只做一个运算。（类似编译的效果）

```
z = SQRT( ADD(MUL(x, x), MUL(y, y)) )
```

```
var r0, r1          // 临时数据 寄存器

r0 = MUL(x, x)
r1 = MUL(y, y)
r0 = ADD(r0, r1)
z  = SQRT(r0)
```

总结

之前的混淆方案，都没有改变 流程顺序。

优点

- 性能损耗小

缺点

- 容易分析

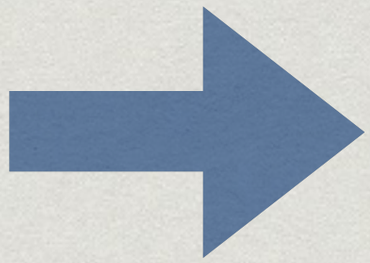
高级混淆

流程混淆

将流程划分成多个 块，通过 状态机 驱动

```
var pwd = prompt()

if (pwd == '1') {
    alert('yes')
} else {
    alert('no')
}
```



```
blocks = [
    () => { pwd = prompt() },
    () => { if (pwd!='1') pc = 3 },
    () => { alert('yes'); pc = 4 },
    () => { alert('no') },
]

pc = 0

while (fn = blocks[pc++]) {
    fn()
}
```

流程混淆

演示：https://codepen.io/anon/pen/RVgajB

对抗点

- 打乱原始流程
- 运行时动态算出 pc 值
- 插入额外逻辑，会影响 pc 值
- 语言层面支持 goto 关键字

虚拟机混淆

使用自定义数据，描述 运算 和 流程

```
do {
    op = read(pc)

    switch (op) {
    case 1:     // add
        reg[x] = ADD(reg[y], reg[z])
    case 2:     // sub
        reg[x] = SUB(reg[y], reg[z])
    case 3:     // jmp
        pc = read(pc);
    ……
}
```

虚拟机混淆

优点

- 攻击者需了解指令细节，才能分析程序逻辑。

缺点

- 解释执行，运行效率低。

演示：https://jsfiddle.net/gqLu0uvm/1/

层次结构

- 上层语言

  使用 JS：移植方便，但很多特性难以实现

  其他语言：可以更好优化，但需要改造业务

- 中间表示

  上层语言 → 中间表示，并对逻辑进行混淆

- 底层指令

  中间表示 → 虚拟机指令

对抗点

- 使用 格式复杂 的指令
- 每次生成 不同格式 的指令，以及相应解释器
- 运行时切换指令集
- 指令也是数据，运行时可 动态修改指令
- 虚拟机自身也需一定保护

性能优化

- 现代浏览器

  asm.js、WebAssembly

- 低版浏览器

  Flash / FlasCC

优点：提高计算性能，可增加更多干扰逻辑

缺点：不支持 JS、DOM 等操作，需通过 FFI

</end>